Titre: Title:	MPEG-2 data encryption for transport over ATM/SONET and its clock generator
Auteur: Author:	Beisong Liu
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Liu, B. (2003). MPEG-2 data encryption for transport over ATM/SONET and its clock generator [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7133/

In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

UNIVERSITÉ DE MONTRÉAL

MPEG-2 DATA ENCRYPTION FOR TRANSPORT OVER ATM/SONET AND ITS CLOCK GENERATOR

BEISONG LIU

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU GRADE DE MAÎTRISE ÈS SCIENCES APPLIQUÉES (M.Sc.A.)

(GÉNIE ÉLECTRIQUE)

Juillet 2003

Your file Votre référence
ISBN: 0-612-86411-1
Our file Notre référence
ISBN: 0-612-86411-1

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

MPEG-2 DATA ENCRYPTION FOR TRANSPORT OVER ATM/SONET AND ITS CLOCK GENERATOR

présenté par: Beisong LIU

en vue de l'obtention du diplôme de: MAÎTRISE ÈS SCIENCES APPLIQUÉES

a été dûment accepté par le jury d'examen constitué de:

M. YVON SAVARIA, Ph.D., Président

M. MOHAMAD SAWAN, Ph.D., Directeur

M. GUY BOIS, Ph.D., Membre

DEDICATION

To my husband, Peijian

my son, Max,

and my parents

ACKNOWLEDGEMENTS

I am grateful to my supervisor, Professor Mohamad Sawan, for his constant confidence, support, encouragement, and guidance throughout this thesis. I am also thankful for the help and friendship of the members of the Polystim research team, who were always ready to engage in useful discussions.

I would like to express my appreciation to Yamu Hu, Mathieu Gagnon, Xiaohong Zhao, Abdelouahab Djemouai, Jean-Charles Voghell and Adel Belhaouane for helping me with various design tools.

I would also like to thank Brent Veitch and Hugh W. Pollitt-Smith, who are support engineers at Canadian Microelectronics Corporation. They answered my questions about this project from RTL description to completed custom design and FPGA designs using Synopsys and Xilinx tools.

I am especially grateful to Craig Piercey and Roger Langlois at Tundra Semiconductor Corporation for his English/ French writing and editorial assistance.

Many thanks to my friends for helping to establish a friendly environment in which to work and for providing moral support during the difficult stages of this work.

Finally, I would like to thank my husband, Peijian Yuan, to whom I owe much for his unending patience, encouragement, and understanding without which I could not have completed this work. Special thanks also to my lovely son Max Yuan, for his understanding and for not providing him with full attention during his growing years.

ABSTRACT

With the ever-increasing growth of data communication, the need for security and confidentiality has become a basic necessity. At the same time, it is expected that when Asynchronous Transfer Mode (ATM) / Synchronous Optical Network (SONET) higher transport frequencies become commonly available, the capacity of telecommunication networks will increase enormously, making media like video-conferencing and videophone a reality, rather than fiction. We propose in this Master thesis a system architecture to support encrypted Moving Pictures Expert Group (MPEG-2) stream transport over ATM/SONET. Some of the key technical issues are discussed and solved. The goals of our work include:

- Develop a new ALL-Digital Delay Locked Loop (ADDLL) function and a Mirror Delay Approach (MDA) to replace the Phase Locked Loop (PLL) technique in order to implement a Multiple-High-Frequency Synchronous Clock Generator (SCG) to support an encrypted MPEG-2 stream transport over an ATM/SONET system.
- Construct an encrypted MPEG-2 stream transport over ATM/SONET system. This system can support three standards (MPEG-2/ATM/SONET). The emerging information and data stream may be encrypted and decrypted at both transmission and reception sides.
- Design of a synchronous dual-port FIFO memory dedicated to the transport of MPEG-2 data toward ATM cells. The implementation of its control unit is achieved using Verilog.

- Implement a Data Encryption Standard (DES) algorithm on an FPGA to validate its operation.

To achieve these goals, the research topics discussed in the following sections represent a summary of our efforts.

First, an All-digital delay locked loop (ADDLL) function is used as a new method to synchronize the input signal and output clocks (19.44 MHz and 155.52 MHz) of the transmission section, as well as timing recovery in the receiver section. Using ADDLL replaces a PLL function which has analog components with a fully digital design. It does not contain any passive components, such as resistors and capacitors. Therefore, the ADDLL overcomes inherent problems associated with analog circuits. The Mirror Delay Approach (MDA) uses the principle of symmetry and proportion to control mirror delay in a multiple-stage frequency generator. This means that the input delay in the phase detector and the input delay in the Frequency Generator are mirrors of each other and symmetric. The variable delay at each stage of the frequency multiplier and the variable delay of the control unit (ADDLL functional block) are proportional. A Multiple-High-Frequency (SCG) using both methods (ADDLL and MDA) is designed. Its frequency output is fixed at 38.88 MHz, 77.76 MHz, and 155.52 MHz from 19.44 MHz. The corresponding chip is fabricated with the CMOS 0.35-μm process from TSMC (Taiwan Semiconductor Manufacturing Company). The test results of the fabricated chip satisfied our expectations. This master thesis allows to prove that the ADDLL function for supporting the synchronization of transmission / reception stages is a valuable precise and

easy to implement method. Also, using an ADDLL and an MDA simplifies the design of a multi-high-frequency SCG.

The second part of this master thesis concerns the encrypted data streams to be sent over ATM cells. In fact, the architecture of an interface that encrypts MPEG-2 streams to ATM cells is based on a synchronous, dual-ported RAM-based FIFO, which is used to access data. Using a dual-ported cell for the FIFO memory makes the control of the FIFO simpler because the read and write controls are non-correlated. Therefore, read and write data to or from the FIFO can occur at the same time without introducing wait states. This technique simplifies the operation of the dedicated finite state machine for this interface.

The third part of our work is the implementation of the data encryption standard (DES) algorithm on FPGA. A key consideration in the implementation of the DES algorithm is that the data stream passes through the encryption block without impacting the transport speed. This DES algorithm's circuit architecture consists of processing the data 16 times through the same stage. The whole process for input, computing, and output data uses a pipeline architecture. The results obtained through VHDL simulations of the encryption DES algorithm are satisfactory. The mapping of the DES algorithm code to a Xilinx FPGA device (XC4028) is achieved. The design's simplicity and fast computation of the DES algorithm offer a dynamic structure for data encryption in data communication systems.

The results obtained during this research work prove that the ADDLL function can be used to easily and quickly construct a delay locked-loop function for timing synchronization on a circuit or system. The ADDLL function implemented is 100 percent

digital. The ADDLL function and MDA together can be used in the transmission section of a data communication system for designing a multiple-stage transmission clock generator. In addition, the ADDLL function may be used for timing recovery in the reception section. Further work is needed to determine more applications that can benefit from this approach.

Further work is also necessary to apply the ADDLL function for timing recovery in a reception, and to implement the whole transmission section on an integrated chip. Also, a more extensive analysis is required for a real system-level simulation, which would include the integration of the clock generator and the remaining part of the system.

RÉSUMÉ

Avec la croissance des communications numériques, les besoins de securité et de confidentialité sont devenus essentiels. De plus, il est possible de prévoir que lorsque les réseaux ATM/SONET deviendront disponibles à grande échelle, la capacité des réseaux de communication suivra une progression très rapide et du même coup, améliorera les moyens de communication tels que les conférences vidéo et les vidéophones. Ce mémoire de maîtrise décrit le diagramme fonctionnel d'un système permettant le transport d'images MPEG-2 encryptées à travers un lien ATM/SONET. Certains autres éléments clés sont également traités dans ce mémoire.

Les objectifs de ce mémoire sont:

- Concevoir un générateur d'horloge capable de produire plusieurs signaux d'horloge synchrones à haute-fréquence d'une façon purement digitale en utilisant le procédé CMOS 0.35 microns de la compagnie TSMC (Taiwan Semiconductor Manufacturing Company).
- Implémenter un algorithme d'encryptage de donneés (Data Encryption Standard - DES) dans un FPGA (Field Programmable Gate Array). L'algorithme est programmé en language VHDL et implémenté sur FPGA Xilinx en se servant de l'outil Synopsys .
- Décrire le code d'une pile FIFO (First In, First Out) synchrone dédiée au transport des données de MPEG-2 à une cellule ATM. Le language utilisé à cette fin est le Verilog.

Les paragraphes suivants sont un sommaire des travaux effectués dans ce mémoire.

Premièrement, un module de génération d'horloge synchrone (GHS) purement numérique est proposé. Il produit un signal dont les plages de fréquence vont de 19.44 MHz jusqu'à

38.88 MHz et de 77.76 MHz jusqu'à 155.52 MHz. Le système fait appel à la technique de boucle de délais de phase entièrement numérique (ALL-Digital Delay Locked Loop - ADDLL) et de l'approche à miroir de délais (Mirror Delay Approach - MDA). La technique ADDLL rend le circuit moins sensible aux variations de température, de tension d'alimentation, et du procédé de fabrication. De ce fait, elle réduit la sensibilité aux variations de fréquence et aux différences de délais provenant du système de distribution du signal d'horloge. Elle est également utilisée pour la synchronisation des données et le recouvrement des délais dans les systèmes de communication sériels. Le MDA est basée sur le principe de proportion et de symétries, afin de contrôler le miroir de délais. Le système utilisant les deux techniques, ADDLL et MDA, permet de produire un signal d'horloge synchrone à fréquences multiples. Le circuit intégré fut fabriqué en CMOS 0.35-um de la compagnie TSMC (Taiwan Semiconductor Manufacturing Company).

En second lieu, nous procédons au design d'un module dédié au transfert de trames encryptées MPEG-2 vers une interface ATM. L'utilisation d'une pile FIFO implementée à l'aide d'une mémoire RAM à deux ports est adoptée, l'avantage vient du fait que les accès aux deux ports de la mémoire sont asynchrones entre eux et donc indépendants. L'écriture peut donc être faite par un port donné, pendant que la lecture de la pile se fait par l'autre port, éliminant ainsi les cycles d'attente. Cette méthode simplifie les opérations de la machine à états synchrone qui contrôlent cette interface.

La troisième partie du mémoire concerne l'implementation d'un algorithme d'encryptage sur FPGA. En ce qui à trait à l'encryptage par l'algorithme DES (data encryption

standard) sur FPGA, le système est basé sur l'idée d'un traitement des données, sans impact sur la vitesse de tout réseau de transport, par une architecture à un étage, dans laquelle les données sont traitées 16 fois. Le procédé d'entrée, de traitement et de sortie des données utilise une architecture pipelinée. L'utilisation du Xilinx Design Manager (XC4028EX) donne une performance acceptable pour l'implémentation de l'algorithme d'encryption. La simplicité du système et la rapidité de traitement de l'algorithme DES offre une structure dynamique pour l'encryptage dans les systèmes de communication numérique.

Les résultats obtenus dans ce mémoire prouvent que l'utilisation d'un ADDLL et d'une MDA permettent de construire un système à boucle de délais pour la synchronisation des signaux. Nous avons produit un système à boucle de délais de phase entièrement numérique. Un générateur de signaux d'horloge synchrone à multiples haute-fréquences peut être utilisé dans la section de transmission des systèmes de communication, pour la génération du signal d'horloge de transmission. De plus, le ADDLL peut être utilisé pour la synchronisation des données dans la section de réception.

De plus amples travaux seront nécessaires pour appliquer la technique de ADDLLL pour le recouvrement d'une horloge en réception et pour l'implémentation d'une interface entre ce système et une interface PCI ou USB. De plus, une analyse plus approfondie est nécessaire pour l'implémentation du système complet sur une même puce.

ABBREVIATIONS

ADDLL	ALL-Digital Delay Locked Loop
AAL	ATM Adaptation Layer
ASIC	Application Specific Integrated Circuits
ATM	Asynchronous Transfer Mode
B-ISDN	Broadband ISDN
CCITT	International Consultative Committee for Telephones and Telegraphs
CDR	Clock and Data Recovery
CLB	Configurable Logic Blocks
CLP	Cell Loss Priority
CMC	Canadian Microelectronics Corporation
CS	Convergence Sub-layer
DCO	Digital Control Oscillator
DES	Data Encryption Standard
DFED	Differentiator with First Edge Detection
DL	Delay Line
DPLL	Digital Phase Locked Loop
FM	Frequency Multiplier
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GFC	Generic Flow Control
GHS	Générntion d'horloge synchrone

HDTV	High-definition television
HEC	Header Error Control
ICDPMLIU	Synchronous clock generator chip ID
IOB	Input/Output Block
ISDN	Integrated Services Digital Network
ITU	International Telecommunications Union
LAN	Local Area Network
MDA	Mirror Delay Approach
Mbps	Mega bits per second
MPEG	Moving Pictures Expert Group
NRZ	Non-Return-to-Zero
NTSC	National Television System Committee
OC	Optical Carrier
OSC	Oscillator
PCI	Peripheral Component Interface
PD	Phase Detector
PES	Packetized Elementary Streams
PLCP	Physical layer Convergence Protocol
PS	Program Stream
PSTN	Public Switched Telephone Network
RTL	Release to Layout
SAR	Segmentation and Reassembly

SDH	Synchronous Digital Hierarchy
SCG	Synchronous Clock Generator
SONET	Synchronous Optical Network
STS	Synchronous Transport Signal level
TDM	Time-division Multiplexing
TOH	Transport Overhead
TS	Transport Stream
TSMC	Taiwan Semiconductor Manufacturing Company
USB	Universal Serial Bus
USR	Universal Shift Register
VCI	Virtual Channel Identifier
VHDL	Very Hardware Description Language
VPI	Virtual Path Identifier
WAN	Wide Area Network

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

D Simulation and Testing Results

E C++ Codes

CHAPTER 1
INTRODUCTION

1.1 Motivation

In today's world, Internet, videoconferencing, videophone, high-definition television (HDTV), multimedia system, and computer telecommunication networks exist everywhere. Storage or transmission of digital images requires huge amounts of memory and a high bandwidth. Therefore, image compression technology is a very promising application field. At the same time, with the ever-increasing growth of these applications, the need for security and confidentiality has become a basic necessity. This is a reason why MEPG-2 encrypted data streams are used as the source data in such image processing systems.

In the past decade, the applications of ATM/SONET technology have exhibited huge progress. The capacity of telecommunication networks will increase enormously making media such as video-conferencing and videophone a reality. In this case, applications of ATM/SONET technology will remain hot topics in telecommunication due to the expected higher transmission rates. In this master thesis, the hardware architectures to implement interfaces to encrypt MPEG-2 streams in a ATM/SONET framework are discussed.

A telecommunication system normally includes transmission and reception sections. Many questions and issues need to be addressed. How to generate a standard high-frequency synchronous clock in the transmission section? How to implement timing recovery in the reception section? These topics are among the most challenging issues in

telecommunication system design. The most popular solution to the synchronization problem is to use a Phase Locked Loop (PLL) function. A PLL is a mixed-signal (analog-digital) circuit, which contains some passive components. These passive components make its design and implementation, in purely digital dedicated fabrication technology, difficult. We propose a synchronization method using a completely digital circuit to replace the PLL circuit, and using an ADDLL function. The Mirror Delay Approach (MDA) uses the principle of symmetry and proportion to control mirror delay in a multi-stage frequency generator. General ADDLL function without MDA only requires synchronous control phase locking for one stage output. The benefit of using both methods is that synchronous control phase locking for multi-stage output is achieved in a Frequency Multiplier.

1.2 Standards

In order to better understand the MPEG-2 encrypted data transport over ATM/SONET system, the respective standards will be explained in the following sections.

1.2.1 Moving Picture Experts Group – Phase II (MPEG-2)

MPEG is a video compression technology formulated by the Moving Pictures Experts Group, a joint committee of the International Organization for Standardization (ISO). The first MPEG standard, known as MPEG-1, was formalized by the MPEG committee in January 1992.

MPEG-1 compression incorporates both audio and video. For National Television Standards Committee (NTSC) video, MPEG-1 uses the Standard Image Format (SIF) of 352x240 at 30 frames per second. Audio is provided as16-bit data streams, stereo sampled at 44KHz. MPEG data rates are variable, although MPEG-1 was designed to provide VHS video quality, and CD-ROM audio quality at a combined data rate of 1.2 mega bits per second.

By resolution and data rate, MPEG-1 targets primarily the computer and games markets. By contrast, MPEG-2, adopted in the spring of 1994, can handle data rates ranging from 2 to 10 mega-bit per second. MPEG-2 is the core compression technology for DVD, the high-density CD-ROM standard that experts predict will replace VHS tapes as the standard for consumer video. MPEG-3 was dropped as a standard. MPEG-4 is a very low-bit-rate codec targeting videoconferencing, Internet, and other low-bandwidth applications.

The MPEG -2 system specification [29] defines the syntax for a valid MPEG-2 bit-stream. This syntax allows the support of functions that include combining multiple coded streams of video, audio, data and other control into a single data stream. In MPEG-2, the coded video, audio, data and control bits, which are called Elementary Streams, are grouped into Packetized Elementary Streams (PES). Each PES packet can contain a varying number of coded bytes from one or several Elementary Streams. Multiple PES streams can be multiplexed into a single MPEG-2 system stream for storage or transmission. Two kinds of MPEG-2 system streams are defined: Program Stream (PS) and Transport Stream (TS).

The Program Stream is intended for a relatively error-free environment (or where errors are corrected, such as CD-ROM applications). The PS packets can vary in size and be relatively long in length (each packet may be a few thousands bytes). The Transport Stream is suitable for a relatively error-prone environment. The TS packets have a fixed length of 188 bytes (4-byte packet header and 184-byte payload). This length was chosen based on the following considerations:

- Encryption: 184 bytes is a multiple of 8 bytes, which is the block-size of popular encryption algorithms
- ATM adaptation: 8 + (2 x 188) = 8 x 48, which can fit into eight ATM cells (explained later in this master thesis).

Note that the PS and the TS are designed for different applications. It is possible to convert from one to the other; however, one is not a subset or superset of the other. Based on the match of encryption algorithms as description above, the focus of discussion is on the TS (Transport Stream) in this master thesis. A block diagram of broadband audio/visual terminal is shown in Figure 1.1.

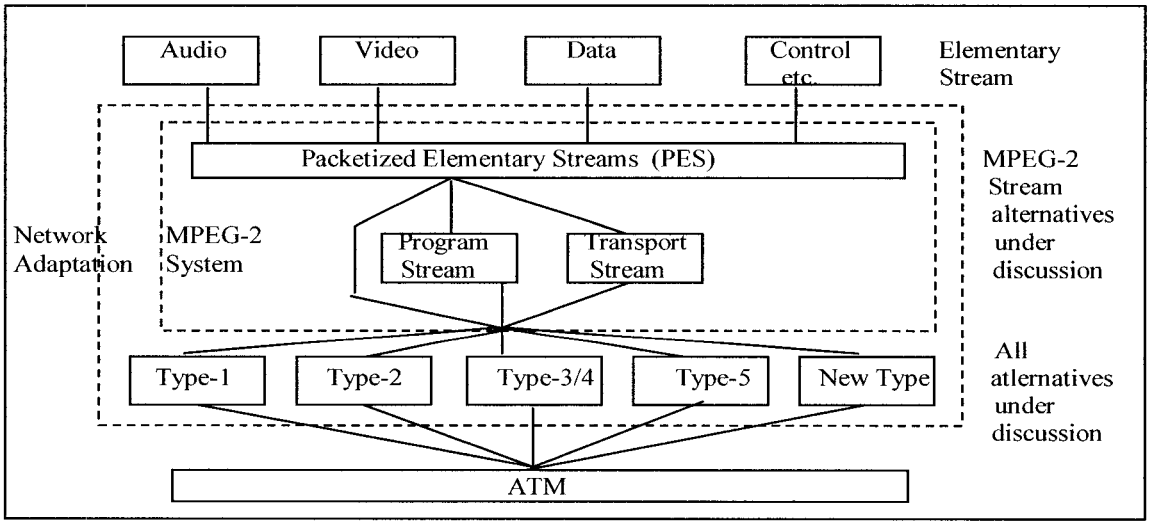


Figure 1.1: Block Diagram of Broadband Audio/Visual Terminal [29]

It depicts an adaptation relationship of data packet from MPEG-2 format to ATM format. The format of Type-1, Type-2, Type-3/4, and Type5 will be explained in the next section.

1.2.2 Asynchronous Transfer Mode (ATM)

ATM is a cell-based technology supported by international standards [4]. ATM is designed to operate over a number of physical layers at data rates ranging from mega bits per second to giga-bit per second. ATM connections allow this operation to offer service from a few bits per second to nearly the giga-bit capacity of the underlying physical layers. An ATM cell consists of 53 bytes; five bytes are the header and the remaining 48 bytes are the payload. The basic structure of an ATM cell is shown in Figure 1.2.

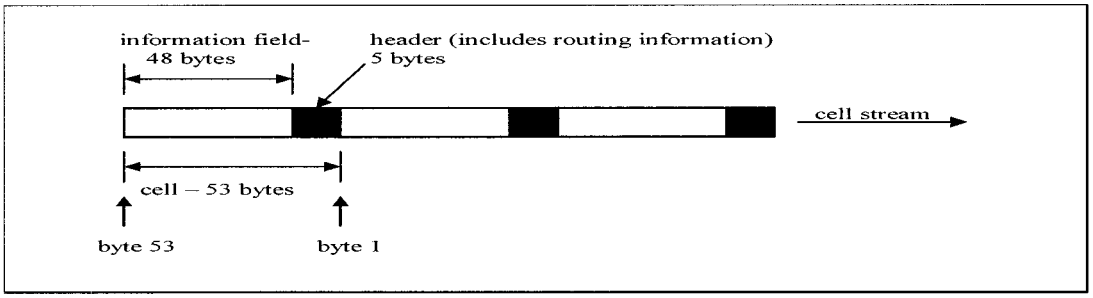


Figure 1.2: Basic ATM Cell Structure [4]

The header appended by the ATM layer is made up of several fields in Figure 1.3

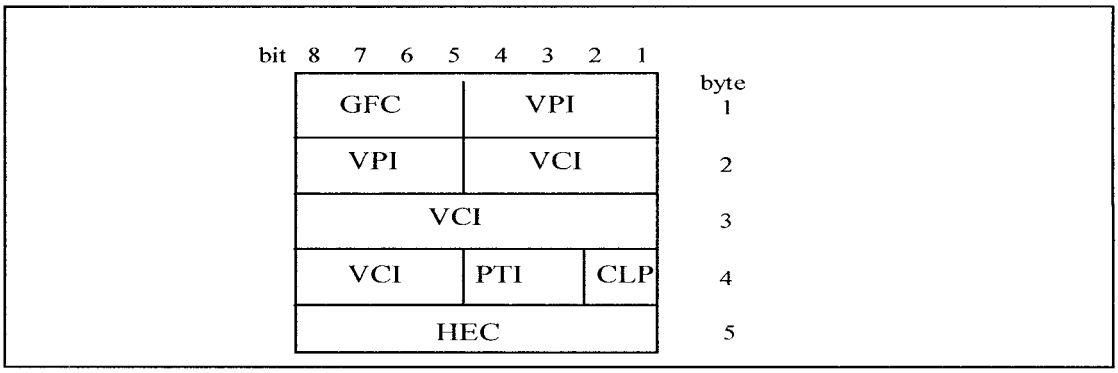


Figure 1.3: ATM Cell Header [4]

The specific functions of the ATM cell fields are elaborated in Table1.1.

Table 1.1: The specific functions of the ATM cell header [4]

GFC	Generic flow control (4 bits) is not defined for a user-network interface (UNI). The value of this field is 0000. For a network interface (NNI), this field is used as a part of the VPI field, providing additional addressing capacity.
VPI	Virtual path identifier (8 bits) allows up to 2^8 = 256 virtual paths. Each VP contains virtual channels. A virtual path is a bundle of virtual channels with the same VPI but different VCIs.
VCI	Virtual channel identifier (16 bits) allows up to 2^{16} = 65,536 virtual channels in one VP.
PTI	Payload type (3 bits) allows ATM to carry up to 2^3 = 8 types of payload. These payload types are identified by the payload type identifier (PTI).
CLP	Cell loss priority (1 bit) is used to determine the eligibility of a cell for discard when the network is congested. If CLP = 1, the cell can be discarded; otherwise, it cannot be discarded.
HEC	Header error control (8 bits) is used for error correction on the other bits in the header. The HEC enables an ATM switch to detect multiple errors and correct single errors.

ATM is a connection-based protocol. Before information transfer, a virtual connection set-up phase is required to reserve the necessary network resources. If sufficient resources are not available, the connection is refused. Each connection is associated with a VCI (Virtual Channel Identifier) and a VPI (Virtual Path Identifier) that are assigned during the call set-up. The VCI and VPI are used in an ATM switch to determine where to relay traffic to the next node. The VPI identifies a group of virtual channel links that

follow the same virtual path. The VCI and VPI have only local significance on the link between ATM nodes and can be changed in the networks. When the connection is released, the VCI and VPI values on the involved links are released, and can be reused by other connections. Figure 1.4 shows the SONET/ATM multiplexing hierarchy. The parameter "In" represents the identifier number of the virtual channels. "Im" represents the identifier number of virtual path. The parameter "nc" represents the number of SONET STS-1 frames (and therefore the line speed). For example, a designation of STS-3 means that the line has 3 times the speed of an STS-1 or 155.52 Mbps. When three STS-1 lines are concatenated such that the frames are phase-aligned and there is a single large payload envelope, it is called an STS-3c line. The parameter "N" represents the number of SONET path.

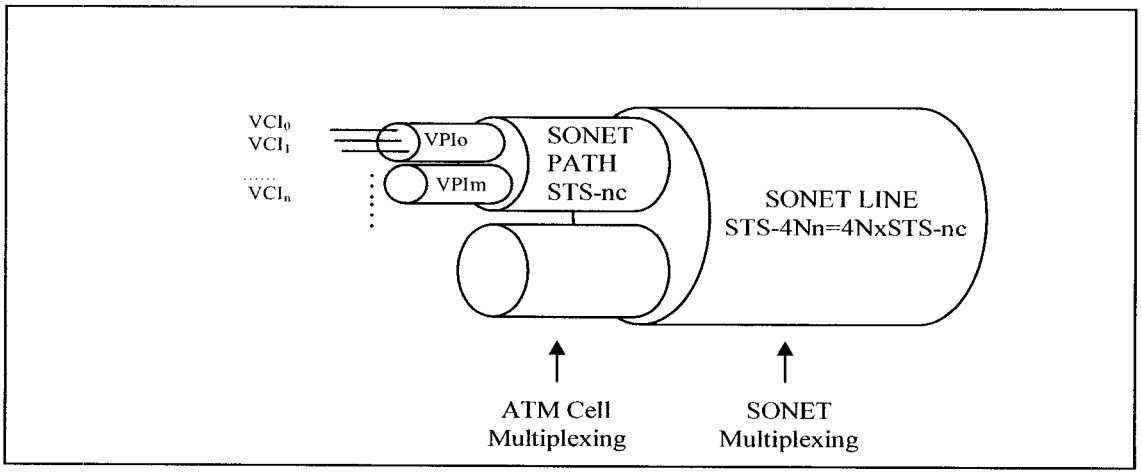


Figure 1.4: SONET/ATM Multiplexing Hierarchy [36]

Although ATM is connection-based, it supports both connection-oriented and connection-less service which allows the transfer of information between service users without the need for end-to-end call establishment procedures. To accommodate various services, four types of ATM Adaptation Layer (AAL) [36], [32] are defined to provide

service specific functions as depicted in Figure 1.5. The selection of AALn/Type-n (n = 1, 2, 3/4 and 5) depends on four factors: data application requirements, connection mode, bit-rates and timing between source and destination. The AAL layer is responsible for acting as the interface between user applications and the ATM cell. The ATM Adaptation Layer (AAL) structure [19] is shown in Figure 1.6.

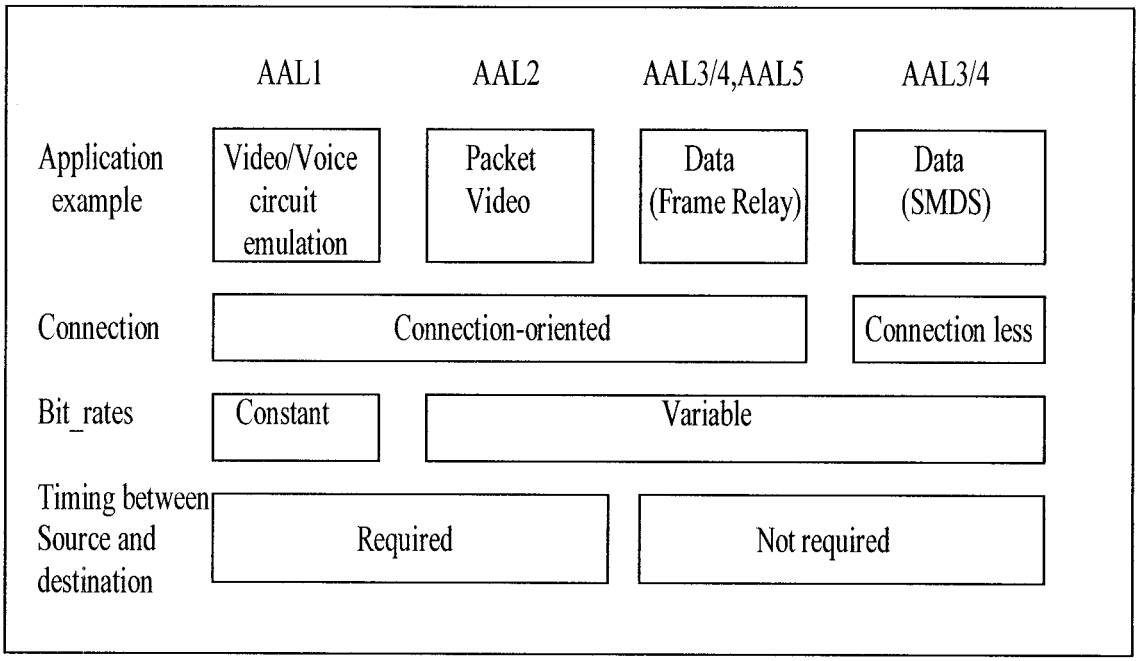


Figure 1.5: ATM Service Classes and AALs [19]

The standard proposes various AALs for different services:

- AAL1 (Type-1): Provides Constant Bit Rate
- AAL2 (Type-2): Provides Variable Bit Rate
- AAL3/4 (Type-3/4): Provides connection-oriented data protocols
- AAL5 (Type-5): Improves AAL3/4. This layer is most common for use in data and compressed video applications.

The AAL is subdivided into two sublayers: the Segmentation and Reassembly sublayer (SAR) and the Convergence Sublayer (CS). The CS depends on the specific service and many support different functions such as clock recovery and data structure recovery. It is divided into two parts. The first is the Service Specific Convergence Sublayer (SSCR). The section part is the Common Part Convergence Sublayer (CPCS). The SAR is responsible for segmentation and reassembly functions. The CS encapsulates the data arriving from upper levels: appends small headers, a CRC (Cyclic Redundancy Check) value, and trailers; and then prepares the data to be segmented in cells.

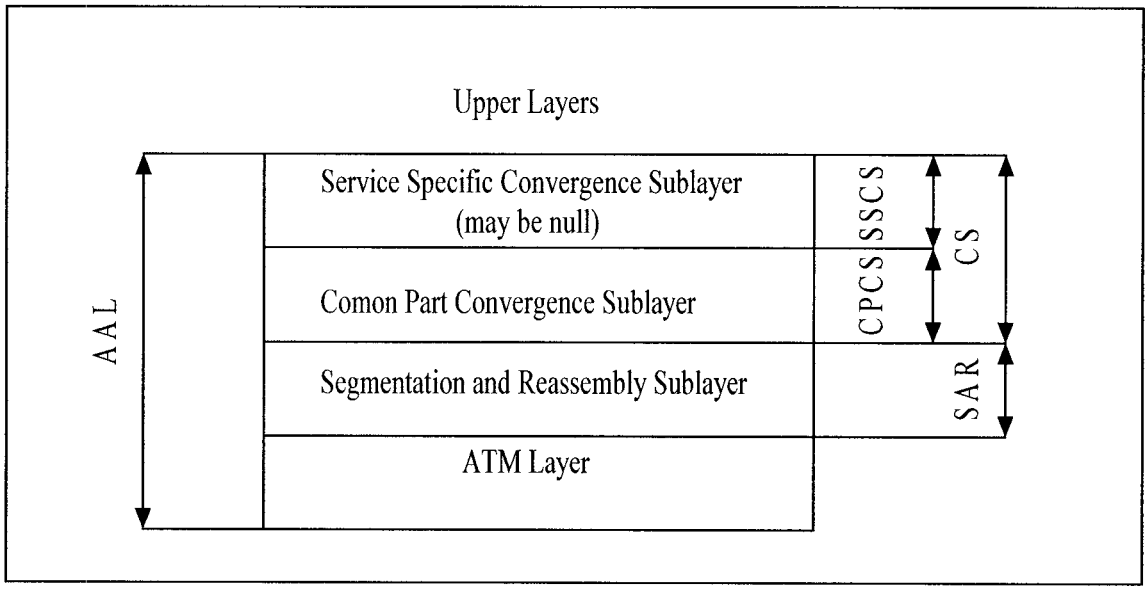


Figure 1.6: ATM Adaptation Layer (AAL)

The SAR is responsible for fragmenting the packets passed from the CS into the ATM layer. Different combinations of SAR and CS sublayers provide different service access points to the layer above the AAL. In some applications, the SAR and/or CS sublayers may be empty.

1.2.3 Synchronous Optical NETwork

Synchronous Optical NETwork (SONET) is a standard for optical telecommunications transport formulated by the Exchange Carriers Standards Association (ECSA) for the American National Standards Institute (ANSI), which sets standards in the U.S. for telecommunications [7]. The comprehensive SONET standard is expected to provide the transport infrastructure for worldwide telecommunications for at least the next two or three decades. The basic SONET frequency is 51.84 Mbps which is referred to as Synchronous Transport Signal level 1 (STS-1). The International Telecommunications Union (ITU) developed the Synchronous Digital Hierarchy (SDH) standard in 1988; SDH is based on SONET. The lowest transmission rate in the SDH standard is 155.52 Mbps; this is referred to as STM-1 (Synchronous Transport Module 1), its transmission rate is three times of STS-1. SONET/SDH standards have equivalent designators among various transmission rates, as shown in Table 1.2.

Table 1.2: SONET and SDH Equivalent Designators [7]

SONET Designation	SDH Designation	Transmission Rate (Mbps)
STS-1	-	51.84
STS-3	STM-1	155.52
STS-9	STM-3	466.46
STS-12	STM-4	622.08
STS-18	STM-6	933.12
STS-24	STM-8	1244.16
STS-36	STM-12	1866.24

The basic structure in SONET is a frame of 810 bytes, which is sent every 125 msec. This allows a single byte within a frame to be part of a 64-Kbps digital voice channel.

Since the minimum frame size is 810 bytes, then the minimum speed at which SONET will operate is 51.84 mega-bit per second as calculated in the equation (1.1):

$$810 \text{ bytes} \times 8000 \text{ frames/sec} \times 8 \text{ (bits)} = 51.84 \text{ Mbps} \qquad (1.1)$$

The basic frame is called Synchronous Transport Signal level 1 (STS-1). It is conceptualized as containing 9 rows of 90 columns each as shown in Figure 1.7:

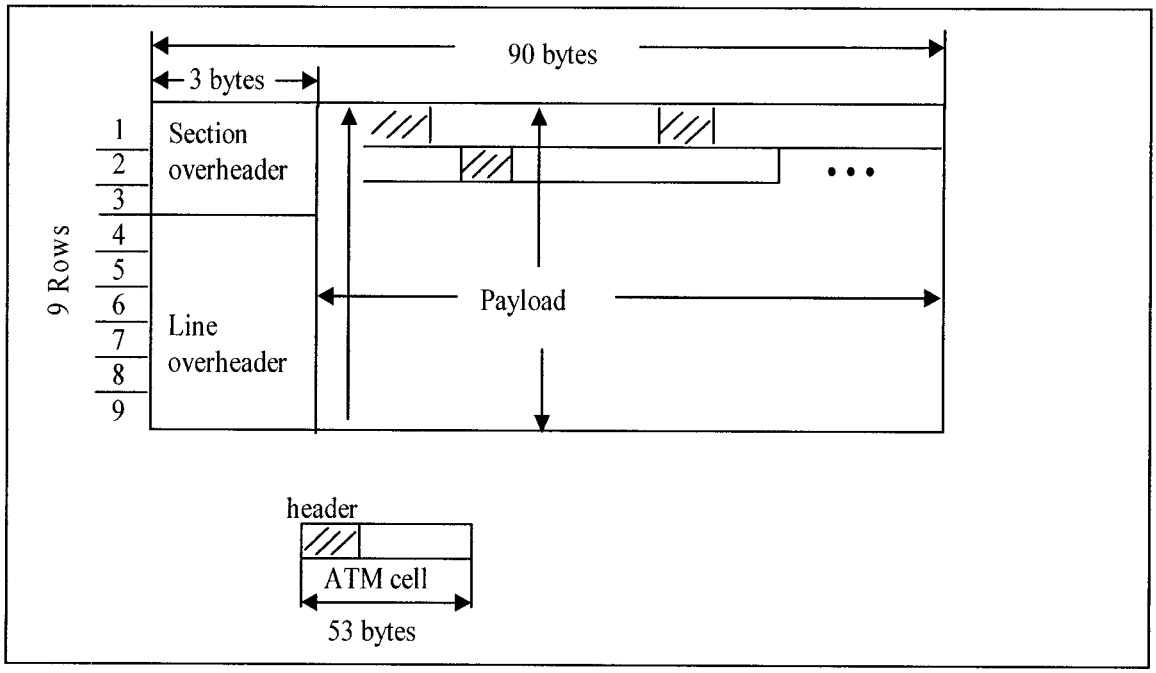


Figure 1.7: STS-1 Frame Structure [7]

- The first three columns of every row are used for administration and control of the multiplexing system. They are called "overhead" in the standard but are very necessary for correct operation of the system.
- "Section overhead" defines the SONET frame and electrical-to-photonic signal conversion. "Line overhead" defines the synchronizing and multiplexing of data into SONET frames.

- The frame is transmitted row by row, from the top left of the frame to the bottom right.

The ITU has defined a worldwide standard called the Synchronous Digital Hierarchy (SDH), which accommodates both SONET and the European line speeds. In this standard a basic frame is defined that is exactly equivalent to (SONET) STS-3c. This frame format has a new name, Synchronous Transport Module level one or STM-1, and has a basic rate (minimum speed) of 155.52 Mbps[1, 25]. This frame format is shown in Figure 1.8.

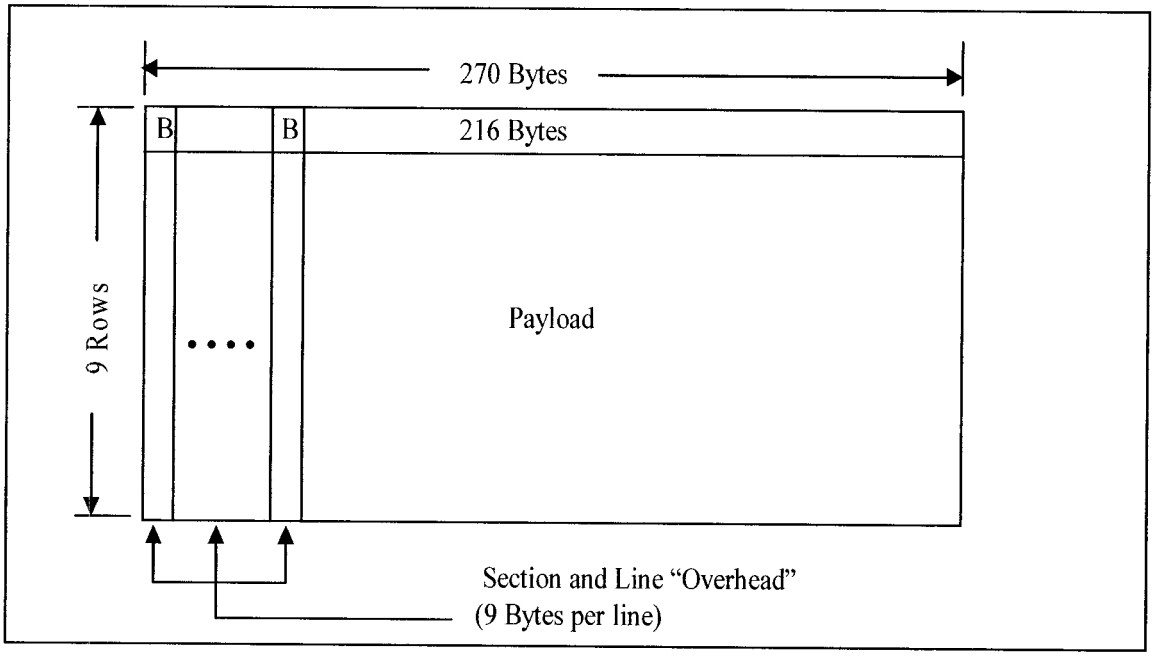


Figure 1.8: STM-1 Basic Frame Format [25]

1.2.4 Data Encryption Standard Algorithm

A complete description of the Data Encryption Standard (DES) [2] is given in the Federal Information Processing Standards Publication 46, published in 1977. This document describes the encryption of two words bit string x of length 64, using a key K of 64 bits

which includes the effective key size of 56 bits; obtaining an encryption (ciphertext) bit string which is again a bit string of length 64. For decryption, the inverse DES algorithm with the same key can be used. Figure 1.9 shows a simplified diagram of the data encryption and decryption system.

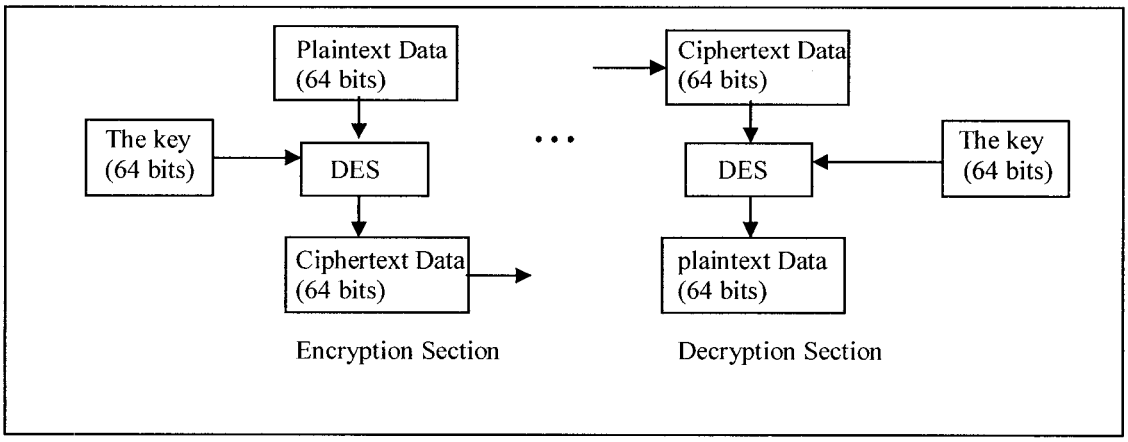


Figure 1.9: Data Encryption and Decryption System [2]

The DES algorithm proceeds in three stages:

- Give two words (64-bit) x, a bit string x_0 is constructed by permuting the bits of x according to an *initial permutation IP* (fixed). We write

$$x_0 = IP(x) = L_0R_0 \tag{1.2}$$

where L_0 comprises the first 32 bits of x_0, and R_0 the last 32 bits.

- Iteration of a certain function 16 times is then computed. We compute L_iR_i, $1 \leq i \leq 16$, according to the following rule:

$$L_i = R_{i-1} \tag{1.3}$$

$$R_i - L_{i-1} \oplus f(R_{i-1}, K_i) \tag{1.4}$$

where: - $\oplus$ denotes the exclusive-or of two bit strings.

- f defines some functions, including Compression Permutation (CP), Expansion Permutation (EP), S-box Substitution (S-box) and P-box Permutation (P-box).

- $K_1, K_2, ..., K_i$ are strings of 48 bits each and are computed as a function of the key K.

- Apply the inverse permutation IP^{-1} to the bit string $R_{16}L_{16}$, obtaining the ciphertext y. That is,

$$y = IP^{-1}(R_{16}L_{16}) \tag{1.5}$$

Note the inverted order of L_{16} and R_{16}.

This algorithm may be divided into 16 identical stages as illustrated in Figure 1.10. DES operates on a 64-bit block of two words. First, an initial permutation is performed, where the block is broken into a right half and a left half: each half is 32-bit long. Then, there are 16 stages of identical operations, called Function f, in which the data are combined with the key. After the sixteenth stage computation is completed, the right and left halves are joined, and a final permutation (the inverse of the initial permutation) finishes off the algorithm.

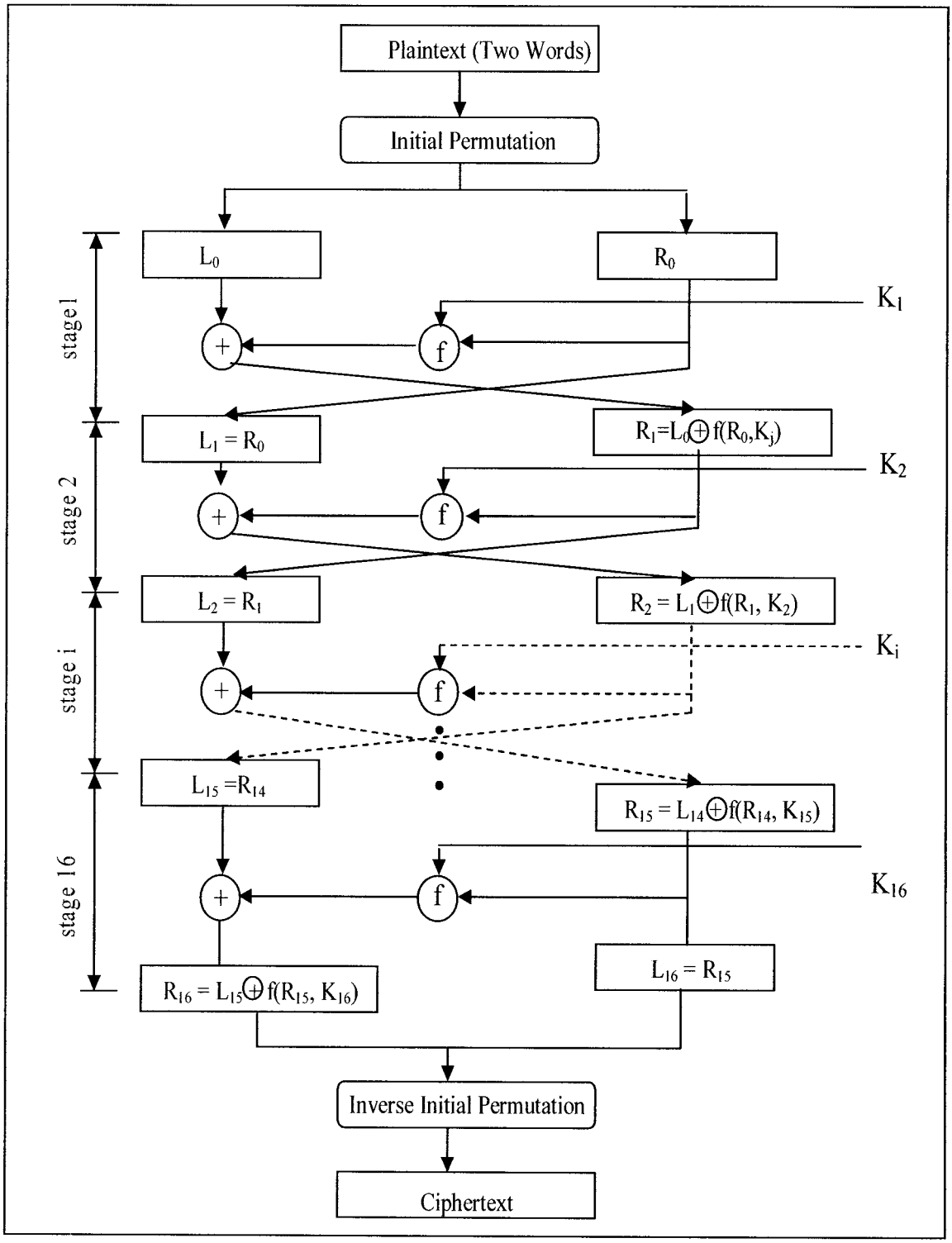


Figure 1.10: Complete representation of the DES operation [2]

In each stage (Figure 1.11), the key bits are shifted (<<), and then 48 bits are selected from the 56 bits of the key (CP). The right half of the data is expanded to 48 bits via an Expansion Permutation (EP), combined with 48 bits of a shifted and permuted key via an XOR. The result is sent through 8 S-boxes producing 32 new bits (S-box), and permuted again (P-box). These four operations (CP, EP, S-box and P-box) make up Function *f*. The output of Function *f* is then combined with the left half via another XOR. The result of these operations becomes the new right half; the old right half becomes the new left half. These operations are repeated 16 times, comprising 16 stages of computations.

An S-box is a substitution operation in which six bits of data is replaced by four bits, see American National Standards Institute (ANSI) Inc., "American National Standard - Data Encryption Algorithm [2]. The 48-bit input is divided into eight 6-bit blocks, identified as $B_1 B_2 ... B_8$; block B_j is operated on by S-box $_{Sj}$, as shown in Figure 1.12. The data result of S-box substitution operation is 32 bits.

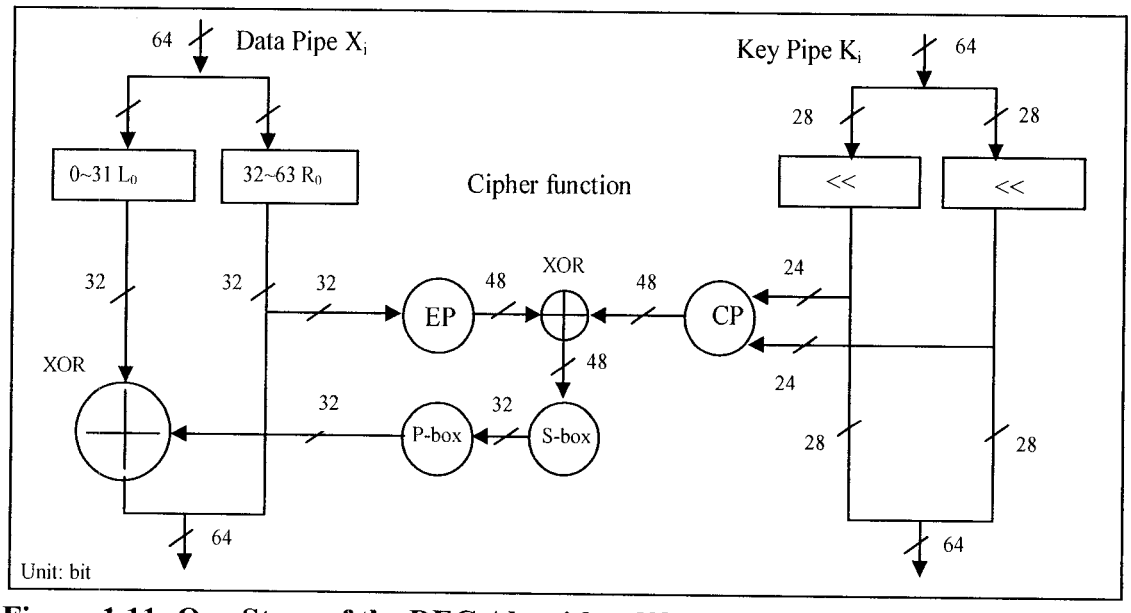


Figure 1.11: One Stage of the DEC Algorithm [2]

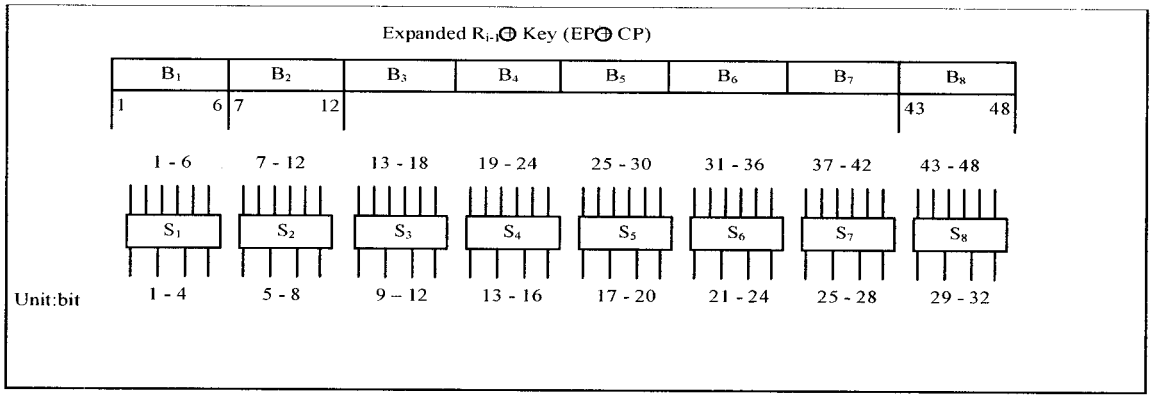


Figure 1.12: S-Boxes Operation on Eight 6-bit Blocks [2]

1.3 Contributions of this Master Thesis

This thesis examines several technical issues to ensure the performance of transport encrypted MPEG-2 video/audio data over an ATM/SONET system. A representation of our research areas is shown in Figure 1.13 by the shaded blocks. The other blocks are not included in our present study. Table1.3 summarizes our contributions in this master thesis.

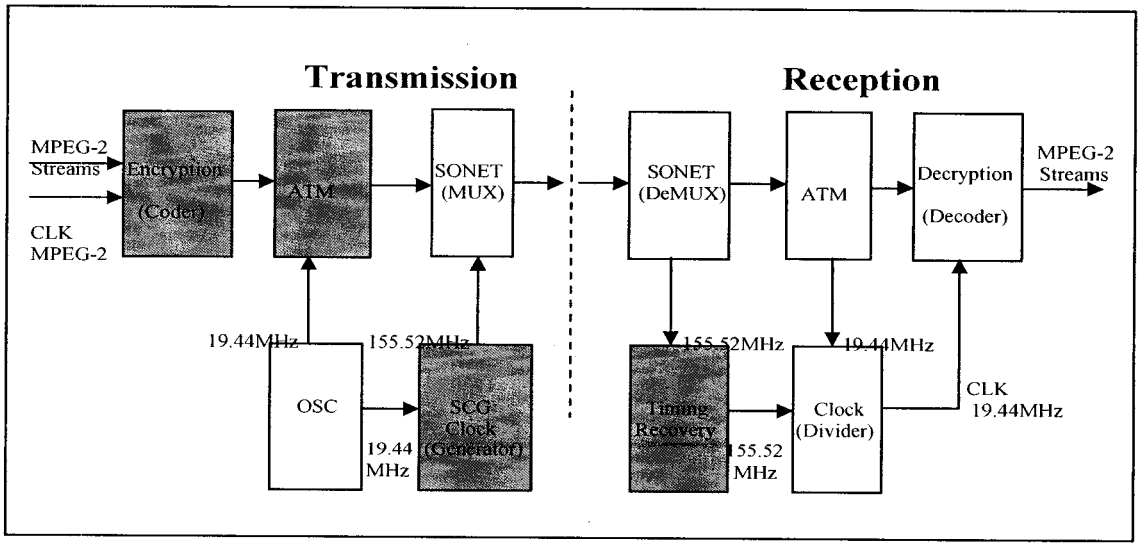


Figure 1.13: The Block Diagram Representation of Our Research Areas

Table 1.3:The Summary of Our Contributions

Blocks Name	Methods	Completed Work
SCG	ADDLL and MDA	Prototype Chip
Timing Recovery	ADDLL	Design and Analysis
MPEG-2 to ATM/SONET	Standards (MPEG-2, ATM, SONET)	1.Architecture design for interface 2. Architecture design FIFO and RTL Coding for FIFO control unit
Encryption Coder	DES	Mapping on FPGA

1.3.1 Transmission Clock Generator and Timing Recovery in the Reception Side

One of the important functions of data transmission over ATM/SONET is to supply the clock information for transmission data. On the reception side, timing recovery is a key function. Currently, the most popular solution of the synchronization problem is to use a Phase Locked Loop (PLL) in the design. In this master thesis, an ALL-Digital Delay Locked Loop (ADDLL) technique is proposed as a new ALL-Digital method to generate the clocks of the transmission section, as well as to implement the timing recovery of the reception section. The benefits of using ADDLL technique are fast phase locking, easy design and implementation, and good stability with ADDLL block. Also, the real system-level integration will be improved due to full digitization synchronous control. A prototype chip of the Multiple-High-frequency Synchronous Clock Generator (SCG) with ADDLL technique and with a Mirror Delay Approach (MDA) is designed and fabricated with a CMOS 0.35-μm process from TSMC (Taiwan Semiconductor Manufacturing

Company). Using MDA allows a system to be implemented which is multi-stage synchronous; that is the system uses multiple synchronous clocks on a chip.

1.3.2 Interface for Encrypted MPEG-2 Streams to ATM Cell

The architecture level design of the interface for the conversion of encrypted MPEG-2 streams to an ATM Cell is also described. The content of this interface include the following key building blocks:

- A FIFO Memory
- Control logic unit based on a Finite State Machine (FSM)
- Packet Information Generator
- Shift registers, counters and other registers.

This interface simplifies the implementation of functions dedicated to mapping the encrypted MPEG-2 streams to the ATM cells. A Multiple-High-frequency SCG is used to improve the performance of this interface. A synchronous, dual-ported RAM-based FIFO is designed. That allows simultaneous writes and reads of data. The FIFO allows data to be read and written separately, which allows the control logic to be simplified. A dual-ported cell simplifies the FIFO control logic because the reading and writing controls do not have to be correlated. Write and read operations to or from the FIFO can occur without inserting wait states. Also, the depth of the designed FIFO is 376 bytes, which is enough storage space for two complete MPEG-2 Transport Stream (TS) packets. For this FIFO, a diagram of the design, as well as the RTL code for the design of the control unit

of the FIFO in Verilog are provided. The Verilog language is used so the design can be consistent with the support environment.

1.3.3 Data Encryption Standard (DES)

An efficient realization of the Data Encryption Standard (DES) algorithm using FPGA technology is proposed. This design approach is based on the idea of streamlining the data without impacting the speed of operation of any network transport. The system architecture consists of iterating the data through one stage block 16 times. The advantage of the proposed design is its hardware-efficiency and faster computation of the DES algorithm, which thereby offers a dynamic structure to access encrypted data by employing a Dual-ported Synchronous FIFO. The following steps have been completed.

- VHDL simulation using Synopsys and Mentor Graphic tools
- Logic synthesis using Synopsys tools
- Mapping to an FPGA device using Xilinx Design Manager tool

The implementation of these steps was done according to the Canadian Microelectronics Corporation's (CMC) tutorial, "Digital Logic Synthesis Using Synopsys and Xilinx".

1.4 Organization of the Master Thesis

This master thesis is composed of seven chapters. The current chapter present an introduction to the topic of the project. Definitions of main issues are given. The Moving Picture Experts Group - Phase 2 (MPEG-2), Asynchronous Transfer Mode (ATM), Synchronous Optical NETwork (SONET) and Data Encryption Standard Algorithm

(DES) are described. The motivation is then discussed and the work contributions are described.

In chapter 2, a literature review for main topics related to this project is provided.

In chapter 3, the implementation of the physical interface for the transmission of the encrypted MPEG-2 streams by ATM cells is discussed. A synchronous, dual-ported RAM-based FIFO is the focus of discussion.

In chapter 4, All-Digital Delay Locked Loop (ADDLL) functional features, as well as its application for clock generator and timing recover are described. A prototype chip design Multiple-High-Frequency Synchronous Clock Generator (SCG) with an ADDLL function with an MDA is presented and discussed. The results prove that using the ADDLL function with an MDA to support the synchronization of a multiple-stage FM are promising new techniques for implementing a data transmission system.

In chapter 5, the realization of the Data Encryption Standard (DES) algorithm using FPGA technology is discussed.

In chapter 6, the simulation results for the RTL code, the FPGA design and final chip testing are reported and analysed.

Conclusions and future directions of this research are presented in chapter 7.

CHAPTER 2
LITERATURE REVIEW AND PROSPECTS

In this chapter, research related to our project during the last ten years is introduced. The chapter is divided into four sections. Two sections are historic a survey for research and development of All-Digital Phase-Locked Loops (ADPLL) technology. A third section is a literature review of ATM/SONET interface. Also, a related hot research topic about All-Digital Delay-Locked Loops (ADDLL) technology is explained in the final section.

2.1 Survey of Analog and Digital Phase-Locked Loops

Phase-locked loops (PLLs) have been in use for a long time. The key concept is to use a local oscillator to lock or track input signals in both phase and frequency [38]. In communication applications, a PLL can make a local clock synchronous with another reference signal. Because digital designs are popular, and they fit well with a baseband design, digital PLLs (DPLLs) are widely used in many modern applications, such as ethernet, asynchronous transfer mode, wireless local area network, microprocessors, digital signal processors, etc.

In the last ten years, many PLL/DPLL based designs have been developed. In [15], a non-return-to-zero (NRZ) timing recovery with a digital phase detector (PD), analog low-pass filter (LF), and voltage-controlled oscillator (VCO) are introduced for band-limited applications. In [28], a fully integrated CMOS frequency synthesizer is presented with an analog LF and current-controlled ring oscillator. [41] presents several design techniques

to improve the performance of a PLL with an analog LF and current steering amplifier ring oscillator. These techniques are usually based on conventional design techniques. In [8], the idea of implementing a variable-bandwidth DPLL by finite state machine (FSM) is described and simulated. This design can be completed by using full digital implementation, however a fully custom layout was still to be performed. In [23], ADPLL technology with Digital Control Oscillator (DCO) is proposed. Its characteristics are fast frequency locking, easy design, implementation and good stability.

Recently, a new algorithm for ADPLL with fast acquisition and fast pull-in range is presented in [24]. It is different from a conventional DPLL implementation in the tracking and locking mechanism. In the meantime, the development of ADDLL technology as an extension of ADPLL technology was a hot research topic.

2.2 Review of ADPLL technology

This review is limited to two representative designs [24,11] in order to focus on our research project:

- Portable Design of an ADPLL-Based Clock Recovery Circuit: its main feature is that it can be developed using a hardware description language (HDL) to reduce the design time as well as to improve the system-level integration simulation.
- ADPLL with DCO Hardware Circuit: its main feature is that it uses a switch-tuning DCO to replace the classical VCO but has only half of the hardware cost.

2.2.1 Portable Design of ADPLL-Based Clock Recovery Circuit

The authors of [24] describe an ADPLL based clock recovery circuit. It has already met the system requirements of large pulling range, short lock-in time, high data rate, high operating frequency, and portability. In Figure 2.1, the major functional blocks of the clock recovery circuit are shown.

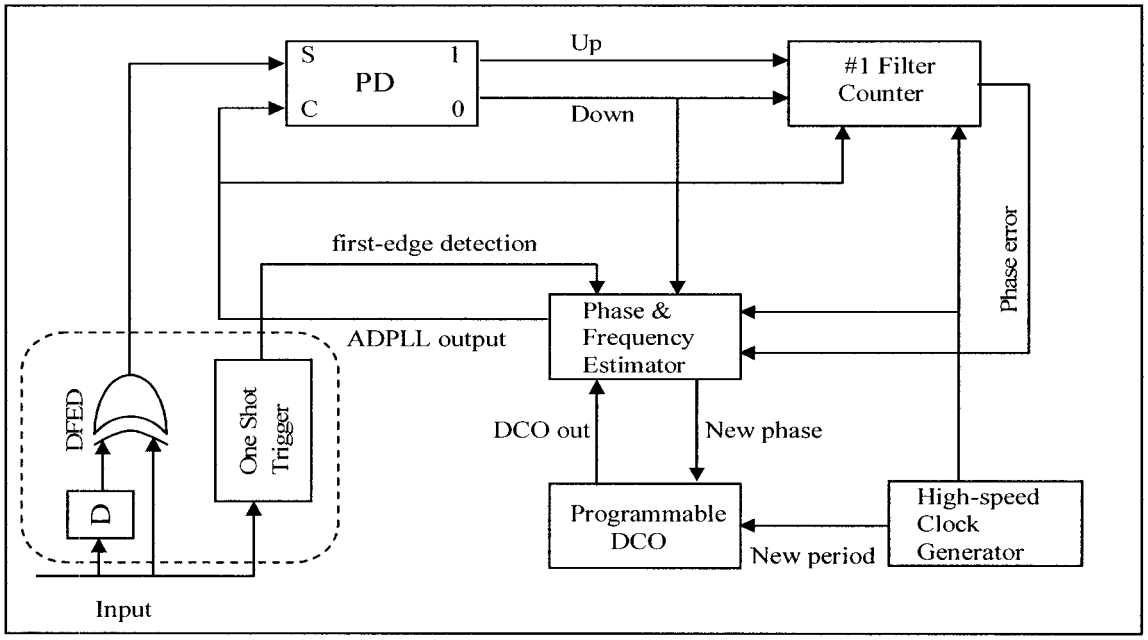


Figure 2.1: Block Diagram of the Cell-based ADPLL Architecture [24]

This ADPLL circuit contains six modules:

1. Differentiator with first edge detection (DFED), whose function is to generate differential pulses to increase resolution and detect the first edge from the input signals to meet synchronous conditions;
2. Phase detector (PD), which is used to indicate phase differences between differential pulses and the digital-controlled oscillator (DCO) output;

3. Up/Down filter counter, which is used to determine the phase errors of two signals by referring to a high-speed clock;
4. Phase and frequency estimator, which is used to measure new phases and frequencies based on original information and new phase errors;
5. Programmable DCO, that is used to generate a target signal;
6. High-speed clock generator, which is used as an on-chip reference timing for controlling and synchronizing all modules to avoid timing errors.

The operation of this architecture is explained as follow (Figure 2.2):

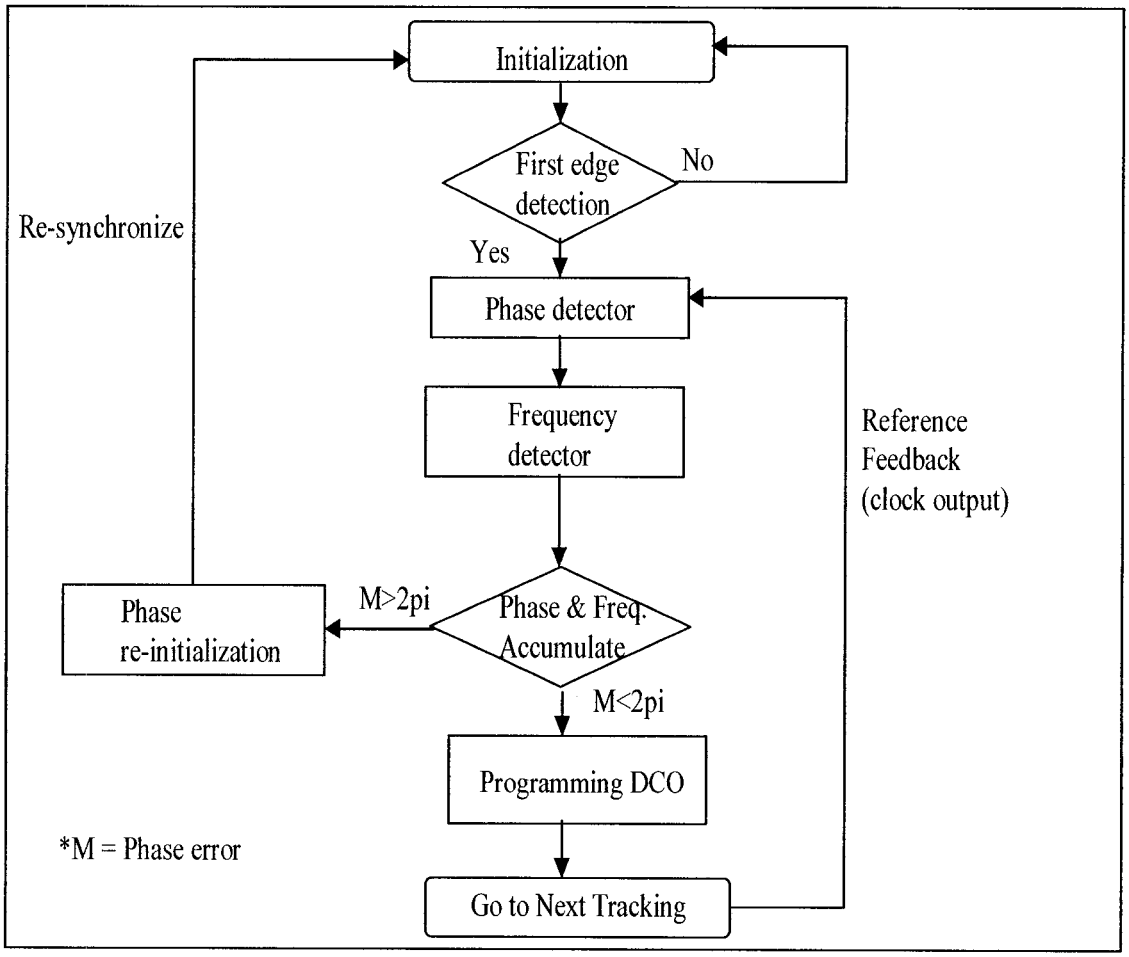


Figure 2.2: Flowchart of the Proposed ADPLL Operation [24]

After the initial settings are loaded, the first valid edge of the input must be detected by the DFED for synchronous operation. By using the first valid edge to trigger all modules for extracting input information, the phase errors (M) are estimated by the filter counter to calculate the difference between the second valid edge of the input and the DCO output. This phase error (M) is used to determine a target period and phase for the DCO. If the phase error is greater than 2 pi, the DCO must be stopped, until the next input edge. This delay corrects the phase error. Finally both correct phase and period (frequency) are determined and fed into the DCO to generate synchronous signals for the next cycle.

These procedures are performed once per valid input transition to ensure that the ADPLL operates with minimum phase and period errors. In a noisy environment, both phase and period of the input signals change randomly, and all noisy phenomena can be classified into two cases. One occurs when the input phase is lagging the DCO output and the other occurs when the input phase is leading the DCO output. In both cases, the ADPLL estimates a new phase and period to minimize errors during input signal tracking. Although the ADPLL becomes stable with any initial setting or any condition, the lock-in time is always proportional to the initial setting.

The operating speed of this architecture is limited by the critical path from the filter counter to the phase and frequency estimator. Note that both the phase and period estimations need one high-speed clock. In addition, input tracking circuit needs at least one high-speed clock. Hence the maximum tracking frequency is equal to $f_{clock}/3$ (without using a differentiator).

The clock recovery mechanism along with a function of frequency synthesizer and an on-chip clock generator [24] are fabricated using a 0.6-μm CMOS process. First, the whole design (including all function modules) is coded in Verilog-HDL using timing information from an in-house target standard-cell library. Then, the Verilog source code is synthesized to generate gate-level netlists and schematics for further simulation and verification.

2.2.2 ADPLL with Simplified DCO Hardware

Generally, there are three types of DCOs. The first one is the "Path delay oscillator", which is designed by cascading many logic gates to form a circular oscillator. The frequency of the path delay oscillator is very limited and can not meet the high frequency requirement. The second DCO category is the "Schmitt-trigger based current-driven oscillator" [15]. It is based on a Schmitt-trigger inverter integrated together with a large capacitor (hundreds of pico farad capacitance) and several control MOS transistors to implement the oscillator. The third category is the "Current-starved ring oscillator". This type of DCO has good linearity and is used in many microprocessor systems. This DCO controls the different MOS switches to get different frequencies. However, the size of this DCO is large and requires a lot of hardware.

In [11], the DCO is a current-starved ring oscillator, but the hardware is smaller. It uses half the layout area of the conventional DCO. This ADPLL is composed of a frequency comparator, a phase detector, a control unit, and the DCO. Figure 2.3 shows the block diagram of the ADPLL.

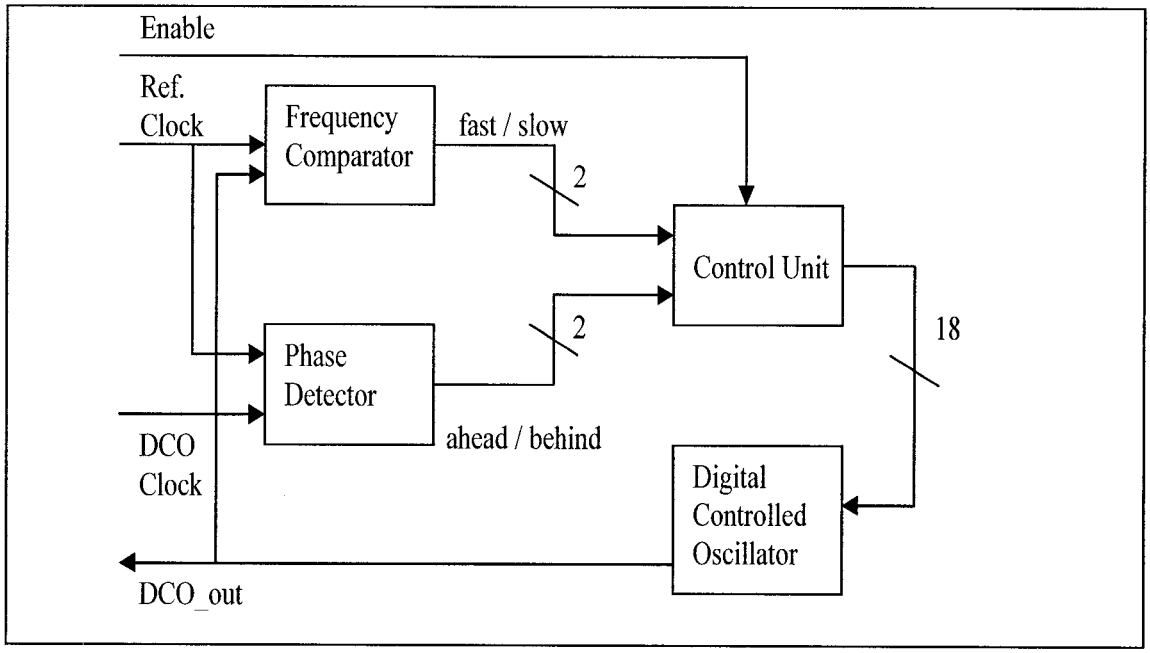


Figure 2.3: ADPLL Block Diagram [11]

The frequency comparator receives the reference clock and the DCO output. The frequency comparator then generates two mutually exclusive "fast" and "slow" signals for the DCO.

The phase detector sets the alignment of the DCO clock edge to the reference clock edge. The phase detector logic block determines "ahead" or "behind" signals. The control unit receives the "fast" and "slow" results of the frequency comparator and the "ahead" and "behind" results of the phase detector as inputs. Mode signals "S[i] (i = 0, 1, 2, and 3)" and control word signals "DCO [0:13]" are output to the DCO. The corresponding simplified circuits are shown in Figure 2.4 and Figure 2.5.

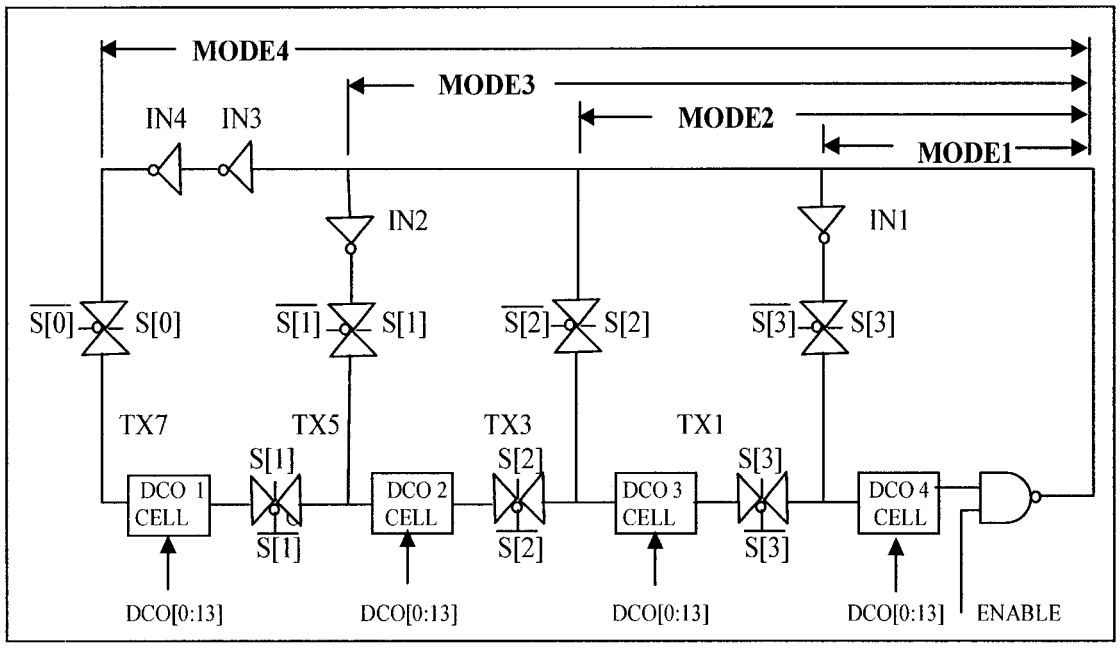


Figure 2.4: Simplified Schematic of a DCO Architecture [11]

The heart of the ADPLL is the DCO. Like most voltage controlled oscillators (VCO), the DCO consists of a frequency-control mechanism with an oscillator block. There are two parameters to modulate the frequency of the ring oscillator. One is the propagation delay time of the inverters, and the other is the total number of the inverters. The conventional DCO tunes the first parameter, but ignores the second one.

The switch-tuning architecture of the DCO described in [11] is shown in Figure 2.4. This DCO is composed of four cells, several transmission gates, balance inverters, and one controlling NAND gate to enable the DCO. In each DCO cell, each inverter is cascaded with 14-bit control MOS devices. The size ratios of the control devices are two times higher than a conventional DCO as shown in Figure 2.5.

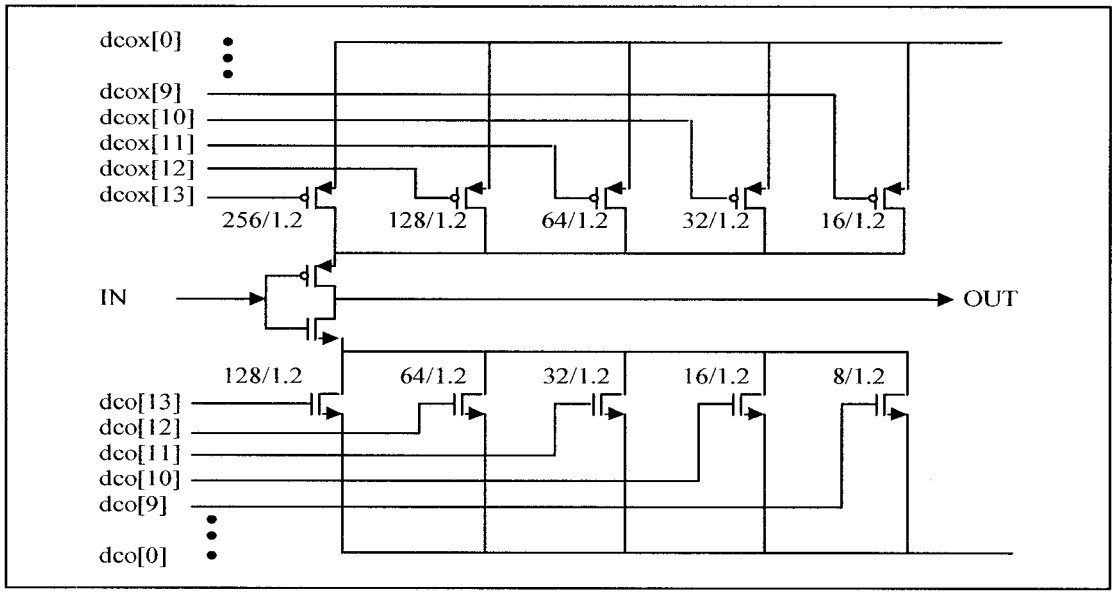


Figure 2.5: Simplified Circuit of a DCO Cell [11]

The most significant control bit (bit 13) corresponds to the largest control device (PMOS transistor), whose dimension (W x L) is 256μm x 1μm. In the DCO, as shown in Figure 2.4, each CMOS transmission gate is inserted to the series with a cell. Therefore, the total number of stages of the ring oscillator can be changed by setting one transmission gate ON and the others OFF. The size of the layout area is reduced significantly compared to the conventional current-starved DCO. This type of DCO [11] has four operating modes. The oscillating periods T_{osci} (i = 1, 2, 3 and 4) of the four modes depend on the following delays times: T_{not} (Inverter delay time), T_{ran} (Transmission gate delay time), T_{dco} (DCO cell delay time) and T_{nan} (NAND gate delay time).

Where,

$$T_{ocs1} = T_{not} + T_{tran} + T_{dco} + T_{nand} \quad (2.1)$$

$$T_{ocs2} = 2T_{tran} + 2T_{dco} + T_{nand} \quad (2.2)$$

$$T_{ocs3} = T_{not} + 3T_{tran} + 3T_{dco} + T_{nand} \quad (2.3)$$

$$T_{ocs4} = 2T_{not} + 4T_{tran} + 4T_{dco} + T_{nand} \quad (2.4)$$

Each operating mode uses a 14-bit control word to set the DCO oscillator frequency. Each mode has a specific frequency range, 60~150, 130~270, 240~357 and 340~430 MHz. The least significant bit resolution of the DCO is 177 ps. The prototype of this 3.3V ADPLL chip is designed and prototyped using TSMC's 0.6 μm CMOS process [11].

2.3 Review of ATM/SONET Interface

Two block diagrams of commercially available ATM/SONET interface are described in this section. In order to focus on designs related to our project, we selected the two most appropriate devices:

- The CYS25G0101DX SONET OC-48 Transceiver (Cypress Semiconductor)
- The S3019 SONET/SDH/ATM OC-3/12 Transceiver (Applied Micro Circuits Corporation)

2.3.1 The CYS25G0101DX SONET OC-48 Transceiver

This transceiver is a building block for a high-speed SONET data communication system. It provides complete parallel-to-serial and serial-to-parallel conversion, clock generation, and clock and data recovery operations in a single chip, optimized for full SONET compliance.

New data is accepted at the 16-bit parallel transmission (TXD [15:0]) interface at a rate of 155.52 MHz. This data is passed to a small integrated FIFO to allow flexible transfer of data between the Input Register and Shifter (Figure 2.6). As each 16-bit word is read

from the FIFO, it is serialized and sent through the high-speed differential Line Driver at a rate of 2.488 Gbits/second.

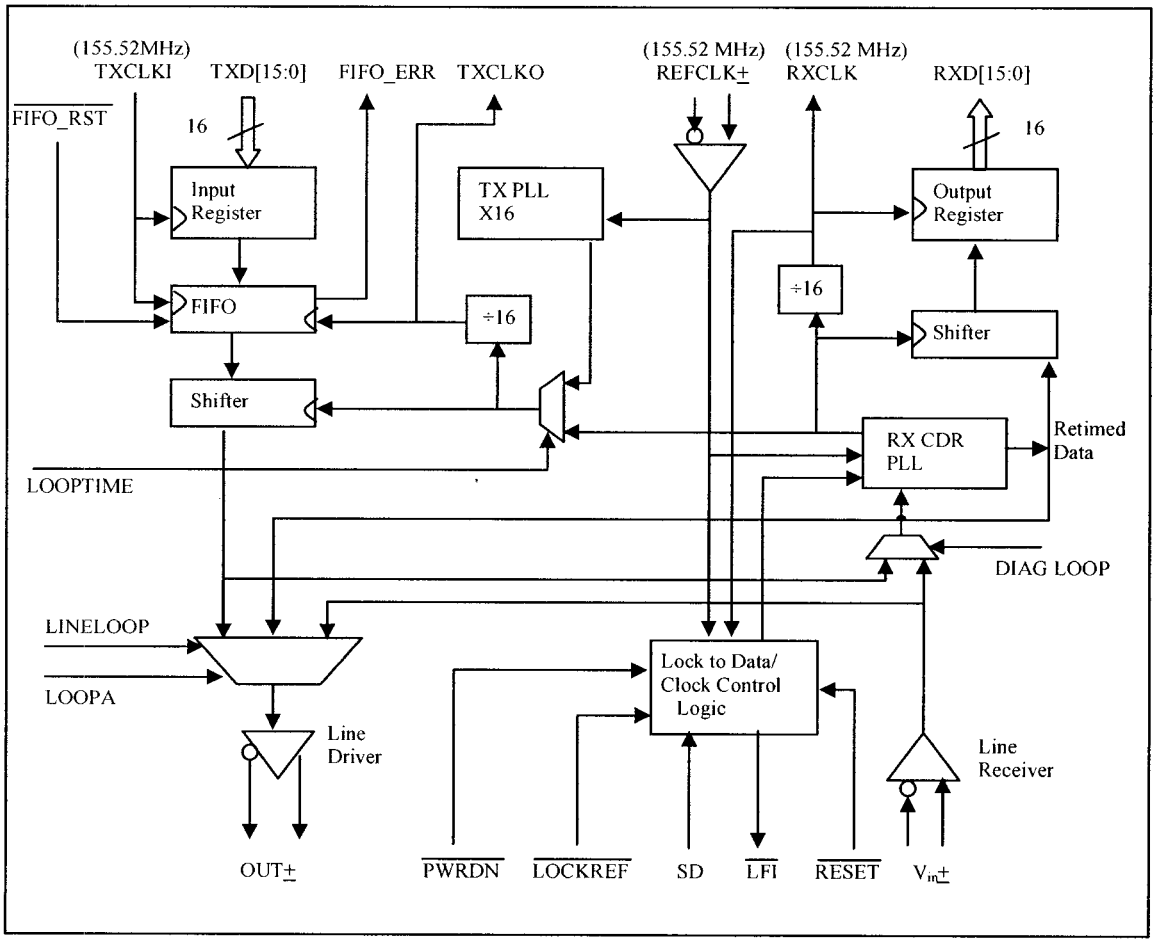


Figure 2.6: Block Diagram of the CYS25G0101DX Transceiver [12]

As serial data is received at the differential Line Receiver (V_{in} ±), it is passed to a clock and data recovery PLL (RX CDR PLL block), which extracts a high precision low-jitter clock from the transitions in the data stream. This bit-rate clock is then used to sample the data stream and receive the data. Every 16 bit-times, a new word is presented at the parallel receiver (RXD [15:0]) interface along with a clock.

The source clock for the transmission data path is selectable from either the recovered clock or an external Building Integrated Timing Source (BITS) reference clock. Multiple loop back and loop-through modes are available for both diagnosis and normal operation. For systems containing redundant SONET rings that are maintained in stand-by mode, the CYS25G0101DX may also be dynamically powered down to conserve system power.

2.3.2 The S3019 SONET/SDH/ATM OC-3/12 Transceiver

This transceiver implements several SONET/SDH serialization/deserialization, transmission, and frame detection/recovery functions on chip. Its block diagram is shown in Figure 2.7. It can be used to implement the front end of SONET equipments, which consists primarily of the serial transmit interface and the serial receive interface. The chip handles all the functions of these two elements, including parallel-to-serial and serial-to-parallel conversion, clock generation and recovery, and system timing. The system timing circuitry consists of management of the data stream, framing, and clock distribution throughout the front end.

At the transmission level, the serializing stage of the S3019 performs the processing of a transmission for a SONET STS-3 or STS-12-bit serial data stream. It converts the 8-bit parallel 19.44 or 77.76 Mbps data stream into bit serial format at 155.52 or 622.08 Mbps. A high-frequency serial data stream can be generated from a 19.44 or 77.76 MHz frequency reference by using an integrated frequency synthesizer consisting of a PLL circuit with a divider in the loop.

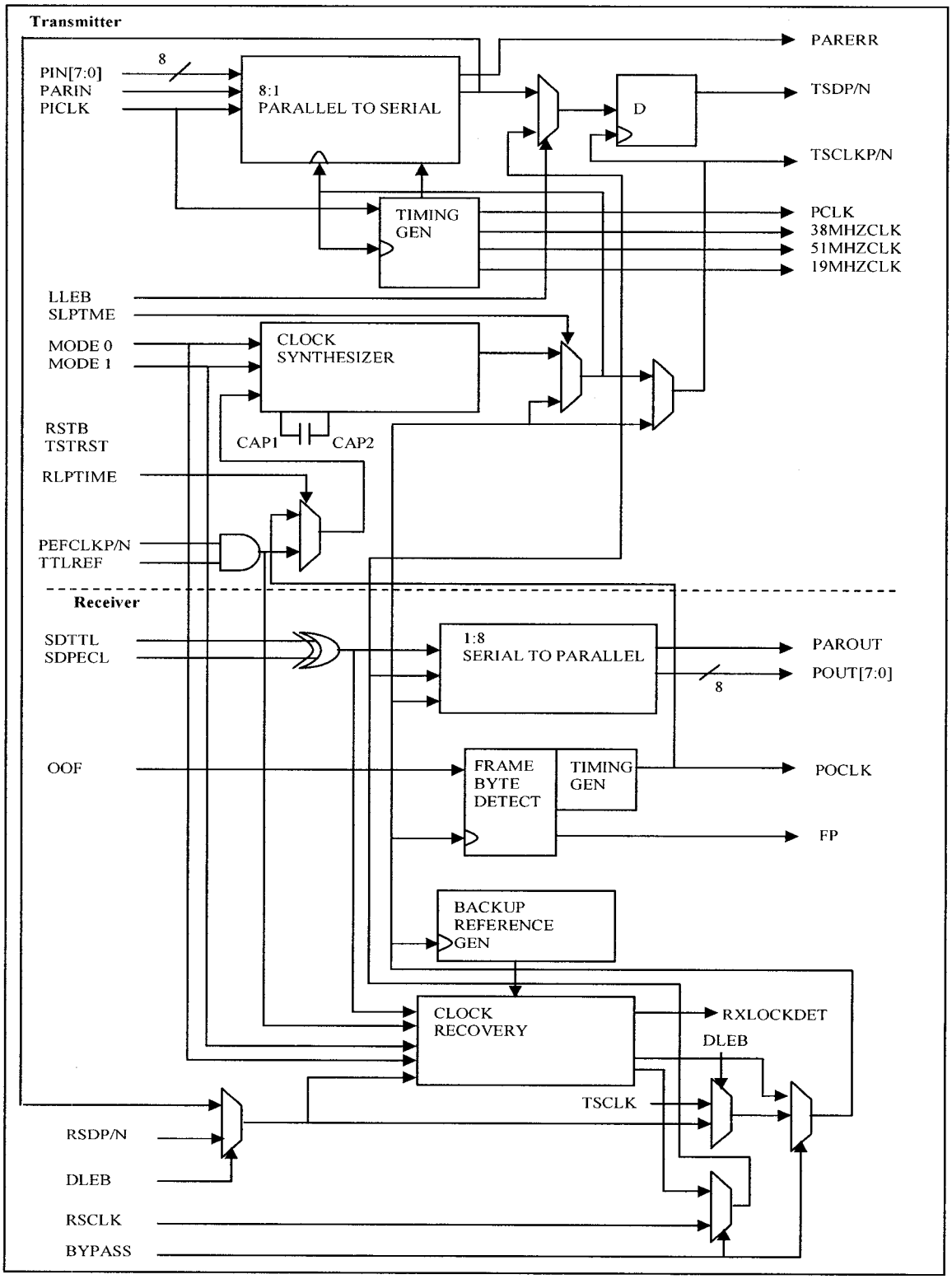


Figure 2.7: Block Diagram of the S3019 Transceiver [3]

This clock synthesizer is a monolithic PLL that generates the serial output clock phase synchronized with the input reference clock. The timing generation function provides a byte rate version of the transmission clock. This circuitry also provides an internally generated load signal, which transfers the data of the PIN [7:0] bus (Figure 2.7) from the parallel input register to the serial shift register.

The parallel-to-serial converter is composed of two byte-wide registers. The first register latches the data from the PIN [7:0] bus on the rising edge of PICLK (Figure 2.7). The second register (labelled on figure 2.7) is a parallel-load shift register which takes its parallel input from the output of the first register.

The S3019 transceiver receives the data and performs the frond-end stage of the digital processing algorithm. It converts the bit-serial 155.52 or 622.08 Mbps data stream into a 19.44 or 77.76 Mbps 8-bit parallel data format. Clock recovery is performed on the incoming scrambled NRZ data stream. A 19.44 or 77.76 MHz reference clock is required to start up the PLL to ensure proper operation under loss of signal conditions. An integral prescaler and PLL circuit is used to increase this reference clock to the nominal bit rate.

The clock recovery block generates a clock that is at the same frequency as the incoming data bit rate at the RSDP/N (Figure 2.7, RSDP/N pin) input, or in loop back the transmitter data output. The clock is phase aligned by a PLL so that it samples the data in the centre of the data eye pattern.

The backup reference generator provides backup reference clock signals to the clock recovery block when the clock recovery block detects a loss of signal condition. It

contains a counter that divides the clock output from the clock recovery block down to the same frequency as the reference clock REFCLKP/N (Figure 2.7, REFCLKP/N pin).

The frame and byte boundary detection circuitry searches the incoming data for SONET frame and byte boundary. Framing pattern detection is enabled and disabled by the Out-Of-Frame (OOF) input. Detection is enabled by a rising edge on OOF, and remains enabled for the duration that OOF is set high. It is disabled when a framing pattern is detected and OOF is reset. When the framing pattern detection circuit is enabled, the framing pattern is used to locate byte and frame boundaries in the incoming data stream (RSD or looped transmitter data). The timing generator block takes the located byte boundary and uses it to block the incoming data stream into bytes for output on the parallel output data bus (Figure 2.7, POUTP/N [7:0] pin). The frame boundary is reported on the Frame Pulse (FP) output when any 48-bit pattern matching the framing pattern is detected on the incoming data stream. When the framing pattern detection circuit is disabled, the byte boundary is frozen to the location found when framing pattern detection circuit was previously enabled. Only framing patterns aligned to the fixed byte boundary are indicated on the FP output.

The Serial to Parallel Converter consists of three 8-bit registers. The first is a serial-in, parallel-out shift register, which performs serial-to-parallel conversion clocked by the clock recovery block. The second is an 8-bit internal holding register, which transfers data from the serial-to-parallel register on byte boundaries as determined by the frame and byte boundary detection block. On the falling edge of the free running POCLK

(Figure 2.7, POCLK pin), the data in the holding register is transferred to an output holding register which drives POUT [7:0] (Figure 2.7, POUT [7:0] pin).

Among the other operating modes, a diagnostic function which is Diagnostic Loopback is used. When the Diagnostic Loop back Enable (DLEB) input is low, a loop back path from the transmitter to the receiver at the serial data rate can be set up for diagnostic purposes. The differential serial output data from the transmitter is routed to the serial-to-parallel block in place of the normal data stream (RSDP/N). SDPECL (Figure 2.7) must be high for diagnostic loop back mode.

2.4 Prospects

For digital very-large-scale-integration circuits, ADDLL function improves over existing techniques to meet the requirements of advanced computer and communication applications. By implementing the design in a hardware description language (HDL), it can be targeted to different CMOS processes through the use of synthesis tools. This process enhances design portability and reduces the design cycle, important issues in digital VLSI designs.

Traditional analog PLL technology is facing new challenges, integrating an analog circuit on a die with digital circuits generates a large amount of digital noise. Using ADDLL function in the design is becoming a popular way to overcome this noise problem. ADDLL offers fast frequency locking time, full digitization and good stability when implemented into a digital system on a chip design. ADDLL technology is maturing rapidly and more solutions using ADDLL may be adopted by industry soon.

Incorporating ADDLL into a design may also improve the system-level integration simulation.

In the past decade, the applications of ATM/SONET technology have exhibited huge progress. The capacity of telecommunication networks will continue to increase enormously, making media like video-conferencing and videophone more popular. In fact, these applications will become more critical due to higher integration levels at high transmission rates. Theses technologies are advantageously combined for image compression and security encryption.

To fit the above requirements for these applications, the architecture for an interface to transport MEPG-2/Encrypted data over ATM/SONET and its clock generator are presented in this master thesis as described below:

- Description of the proposed ADDLL circuit dedicated to implement a multiple-stage clock generator which will support the interface to transport MEPG-2/Encrypted data over ATM/SONET.
- Design of a clock generator using both VHDL and gate-level design techniques. This design is targeted for implementation in standard-cell and custom-definited-cell technologies through the use of standard layout tools.

CHAPTER 3
ENCRYPTED DATA TRANSMISSION OVER ATM/SONET

In this chapter, MPEG-2 encrypted data and its transport over ATM/SONET are discussed.

3.1 Description of the Data Transport Interface

The encrypted data transport interface is divided into two sections: the transmission and the reception. Both sections have corresponding functional blocks such as coding and decoding, multiplexing and demultiplexing, clock generator and timing recovery (Figure 3.1).

The data transport over the transmission section includes functional blocks which:

- Input an MPEG-2 Transport Stream (TS) into an Encryption Coder and output the encrypted data based on a DES algorithm.
- Input encrypted data into an ATM module and subsequently transfer the result to a SONET multiplexing block.
- In a SONET block, the overhead portion of the SONET frame is generated and the output signal is sent as a SONET-3c frame using a 155.52 MHz clock.

At the reception end, all of the main functions of the transmission section are reversed. In addition, in the transmission section, the input data rate of MPEG-2 varies from 2 to 10 Mbps. The output data rate from the SONET (MUX) block of 155.52 Mbps is eight times faster than the output data rate of the ATM block (19.44 MHz). In the reception section, a

timing recovery block is implemented to sample 155.52 MHz frequency from its incoming data and make this frequency synchronous with the transmission frequency in the transmission section. Next, this 155.52 MHz frequency clock signal is divided by 8 in the clock divider block. The resulting 19.44 MHz signal is used as an operation clock on the ATM block and the decryption decoder blocks.

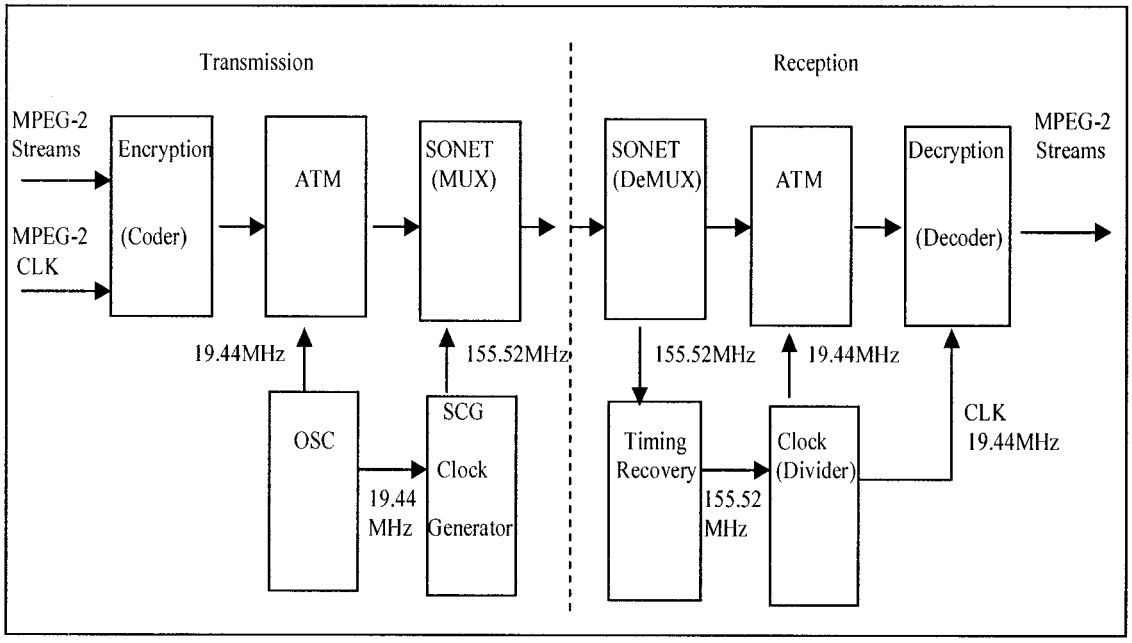


Figure 3.1: Functional Blocks of System

Obviously, the transmission clock generator and the timing recovery block, on the reception side, are key components in the data transport system. If the output signal from the transmission clock generator is unstable or the recovery clock is not synchronous with the transmission clock, then the whole system may be paralysed. Therefore a transmission clock generator is one of most important modules in this work.

3.2 Overview of the Physical Interface

As discussed in [29] and [4], an MEPG-2 TS and an AAL-5 adaptation type are used to encrypt the MPEG-2 streams and before mapping the result into the ATM cells. The data flow for this processing is shown in Figure 3.2.

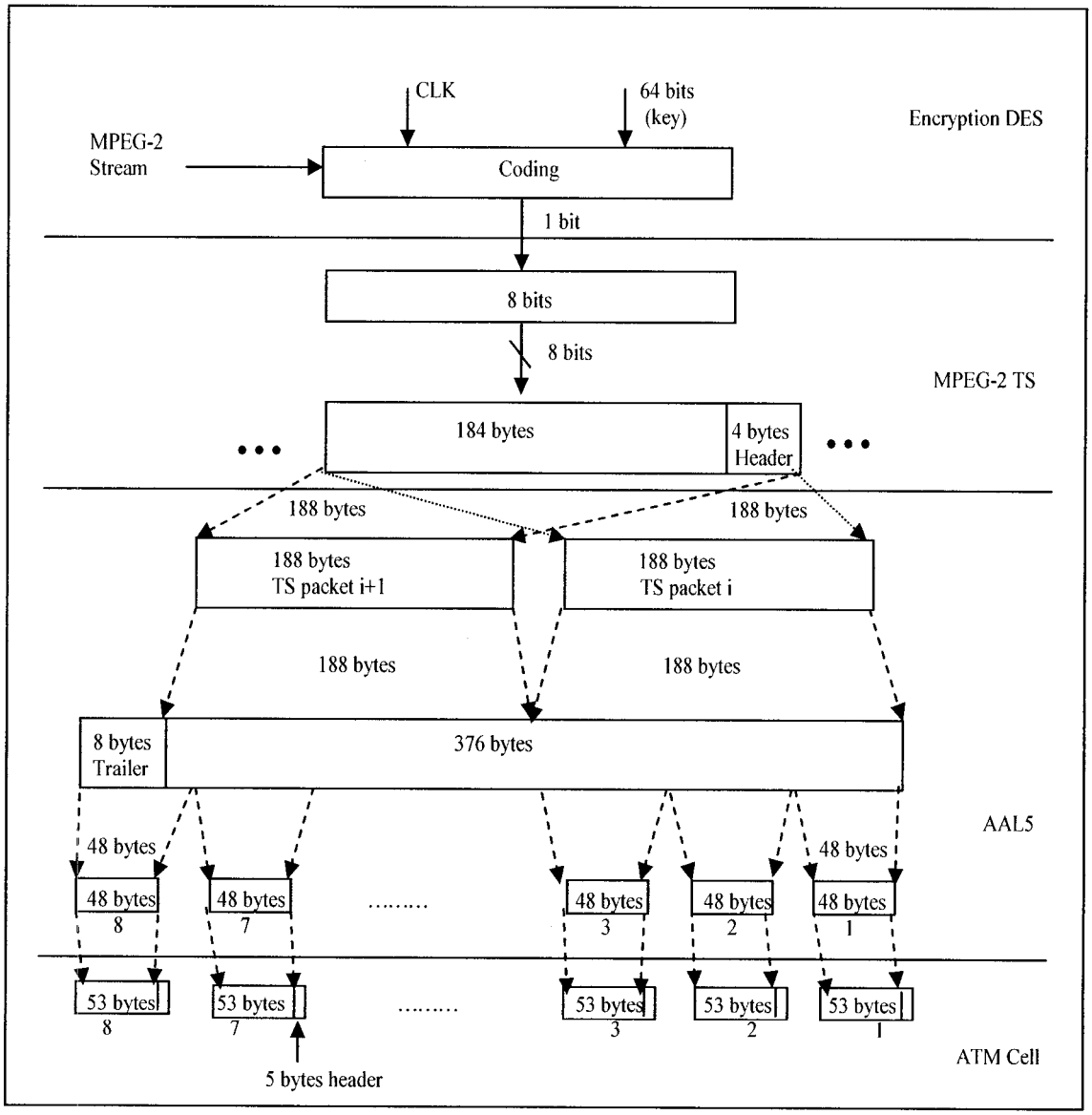


Figure 3.2: Data Flow of Encrypted MPEG-2 TS / AAL-5/ATM Cell

At the encryption level, the MPEG-2 stream is a serial input to the encryption DES encoder and after encryption a serial encrypted output is produced from this encoder. At MPEG-2 TS level, the encrypted data is changed from serial to parallel, then these parallel encrypted data are encapsulated into a packet; each packet is 184 bytes plus a 4-byte wide TS header. Finally, the 188 bytes (184+4) become an MPEG-2 TS packet. In order to be converted to ATM cells, two TS packets are sent to the AAL-5 level. At the AAL-5 level, each set of 376 bytes (two TS packets) add an 8-byte trailer are divided into eight ATM payload cells; each ATM payload cell is 48 bytes wide. At the ATM cell level, a 5-byte ATM header is added to each ATM payload cell.

Based on this data flow, a simple: high throughput architecture for encrypted MPEG-2 stream to ATM cell mapping is considered. A block diagram of this architecture is shown in Figure 3.3.

The architecture of this interface enclose a FIFO memory, four shift-registers and a multiplexer (MUX), six counters, a control logic unit based on a Finite State Machine (FSM), a packet information generator, a series to parallel converter (S/P) and some registers. The important of these modules is shown here:

• **SERIAL TO PARALLEL CONVERTER:** This module is a serial-in, parallel-out shift register. Its 8-bit parallel data output are read by the FIFO at the rising edge of the *flag_in* signal;

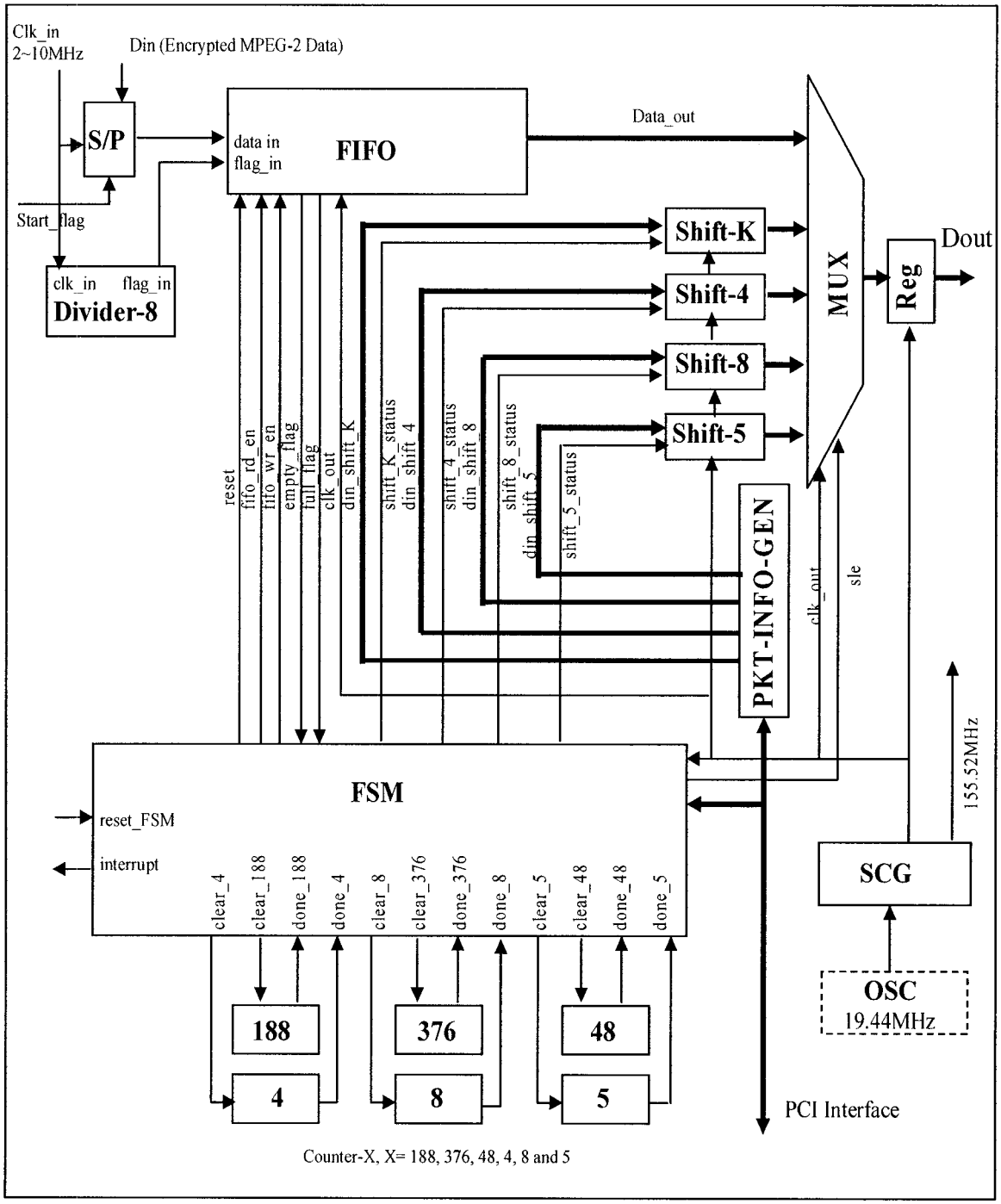


Figure 3.3: Interface Architecture of Encrypted MPEG-2 Stream to ATM Cell Mapping

• **DIVIDER-8:** This block generates the flag signal (*flag_in*) for the clock input of the FIFO. Every eight *clk_in* cycles, a *flag_in* cycle is generated which is 8-times the divider clock cycle;

• **FIFO**: The data rates of input and output signals interfaced with the FIFO are different, the input data rate is slow (2~10 MHz) and the output data rate is fast (19.44 MHz). In order to make the data transfer at the interface produce in a continuous data stream, a temporal store unit FIFO is needed. A data byte is written into the FIFO when the *flag_in* signal is asserted. A data byte is read from the FIFO on the rising edge of the output clock (*clk_out*) if *read_en* signal is set by the FSM.

Another design consideration is the depth of the FIFO. If the depth of the FIFO is large, the FIFO becomes more expensive. Otherwise, if the FIFO depth is reduced the control of the interface is more difficult to implement. Based on Figure 3.2, the longest user data storage happens when the two 184-byte (368-byte) TS packets are combined. The 4-byte TS header, 8-byte trailer and 5-byte ATM header can be stored by the shift registers individually. Therefore, 368 bytes are selected as the depth of the FIFO.

This FIFO buffers the data from the MPEG-2 TS to the ATM cell. A detailed explanation about writing and reading data to/from the FIFO is described in section 3.3;

• **COUNTER:** There are six counters in this interface. Counters 4, 5 and 8 (Figure 3.3) create flag signals individually to signal the FSM. These flags provide status signals (done/undone) of the shifting operations for the 4-byte TS header, the 8-byte AAL-5 trailer, and the 5-byte AT header. The other three counters: 48, 188 and 376 (Figure 3.3) are responsible for counting the output of the FIFO and individually signalling that the

operation has completed successfully for the 188-byte MPEG-2 data, the 376-byte AAL-5 data, and the 48-byte ATM payload data. The *clear_x* (x = 4, 5, 8, 48, 188, and 376) signals are generated by the FSM individually;

• **PKT-INFO-GEN:** A packet information generator creates the information needed to produce the 4-byte TS header, 8-byte AAL-5 trailer, and 5-byte ATM header. All of the packet information is loaded through the external PCI interface and counters. Also, in order to keep the data transfer on the output of this interface in a continuous data stream, some empty bytes (non-information data byte) must be added into the ATM cell. The non-information data byte is also generated by the PKT-INFO-GEN block;

• **SHIFT REGISTER:** Four shift registers are used to store the 4-byte TS header, 8-byte AAL-5 trailer, 5-byte ATM header and empty-data byte individually. When the signal *shift_x_statu* (x = 5, 4, 8 and K) is set to 1 by the FSM, the shift-X (X = 4, 8, 5 and K) register sends the data out of the interface;

• **FSM:** This unit is a key section of this interface. It provides control signals to read data from the FIFO and read packet information from the Shift-X register (X = 4, 8, 5 and K). Figure 3.4 shows the flowchart for the read data controls of the FSM. This flowchart is based on the encryption / MPEG-2 TS/AAL-5 / ATM cell data flow depicted in Figure 3.2. The read control algorithm consists of the following steps:

1. Reset counter-X (X=188, 376, 48, 4, 8, and 5).
2. Set *shift_5_status* signal = 1, counter-5 starts to count, Shift-5 register will output data.

3. The FMS set *shift_4_status* signal = 1, counter-X (X=188, 376 and 48) starts to count when *done_5* = 1 and *done_188* = 0. Shift-4 register will output data. Otherwise, if *done_188* > 0, go to next step (3-D).
4. Set *fifo_rd_en* signal = 1 when the FIFO is not empty, then sent the output data from the FIFO, otherwise send non-information output data from the Shift-K register.
5. If signal *done_48* = 0, go to step (3-D); if *done_188*= 0, go to step (3-D); if *done_376* = 0, go to step (2).
6. If *done_48* = 1, *done_188* = 1 and done_376 = 1: set *shift_8_status* signal = 1, and counter-8 start to count, Shift-8 register will output data.
7. Return to step (1) to reset when *done_8* = 1.

• **SCG:** The output clock of the FIFO, the clock of the FSM, and all of the clocks for the shift registers are supported by a 19.44 MHz clock from the signal SCG. The 155.52 MHz clock is used during a SONET transfer in the next protocol level. Both the 19.44 MHz and 155.52 MHz clocks from SCG are phase locked on a 19.44 MHz external clock.

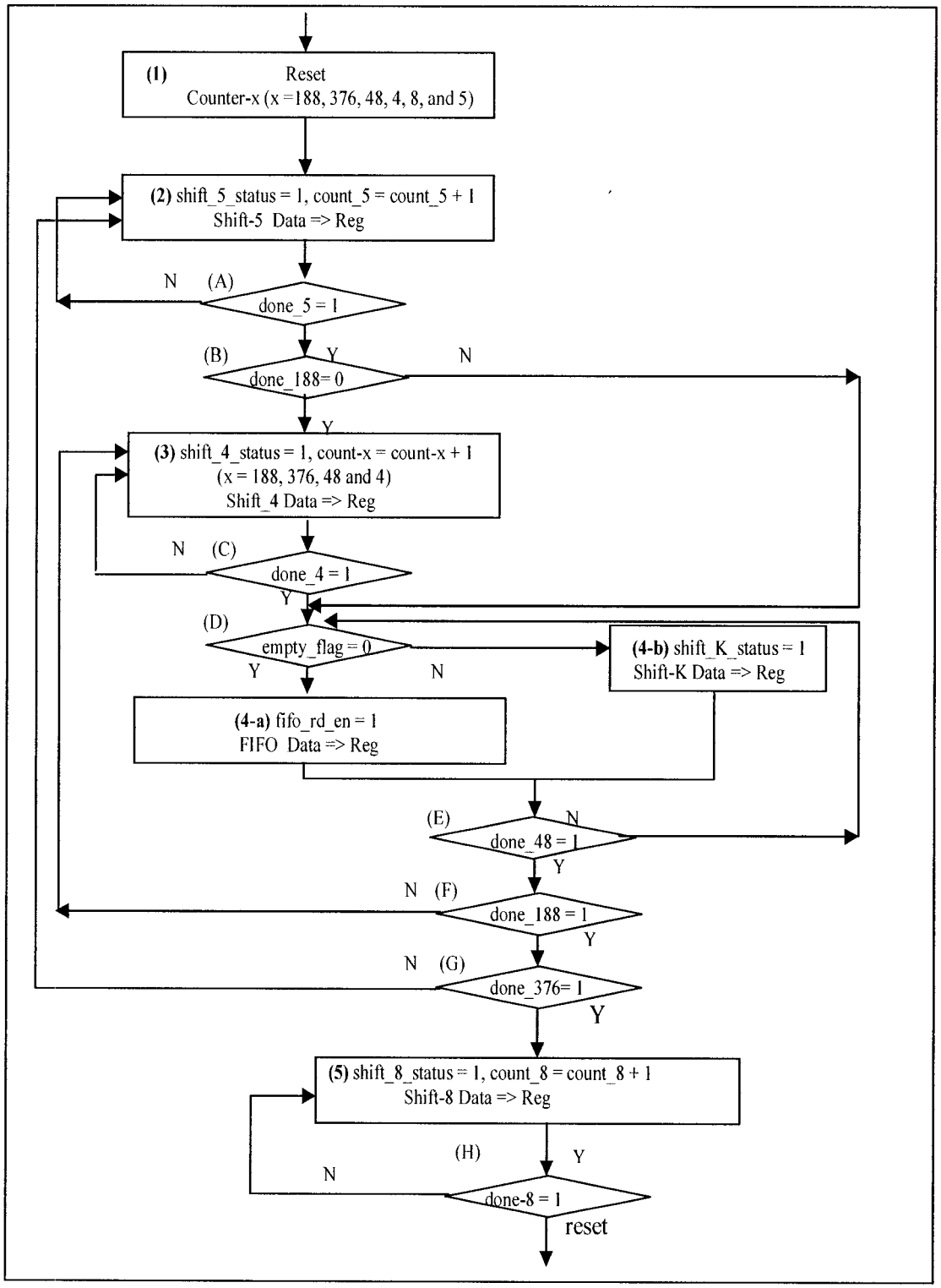


Figure 3.4: Flowchart for the Read Data Controls of the FSM

3.3 FIFO for Data Transport from MPEG-2 TS to ATM Cell

Using FIFO buffers to pass information between different functional blocks with different data rates has been a standard practice in interface design [18, 39]. This section explains the key features of the synchronous FIFO (376 byte depth) as a buffer for interfacing the MPEG-2 TS data stream to an ATM cell. The FIFO is composed of very large buffers based on a static memory array (Figure 3.5). The internal RAM is a dual-ported memory that is addressed by the use of internal pointers. These pointers determine which address of the RAM provides the data during a FIFO READ or stores data during a FIFO WRITE. The different read and write clocks make the data input and the data output of the FIFO independent of each other.

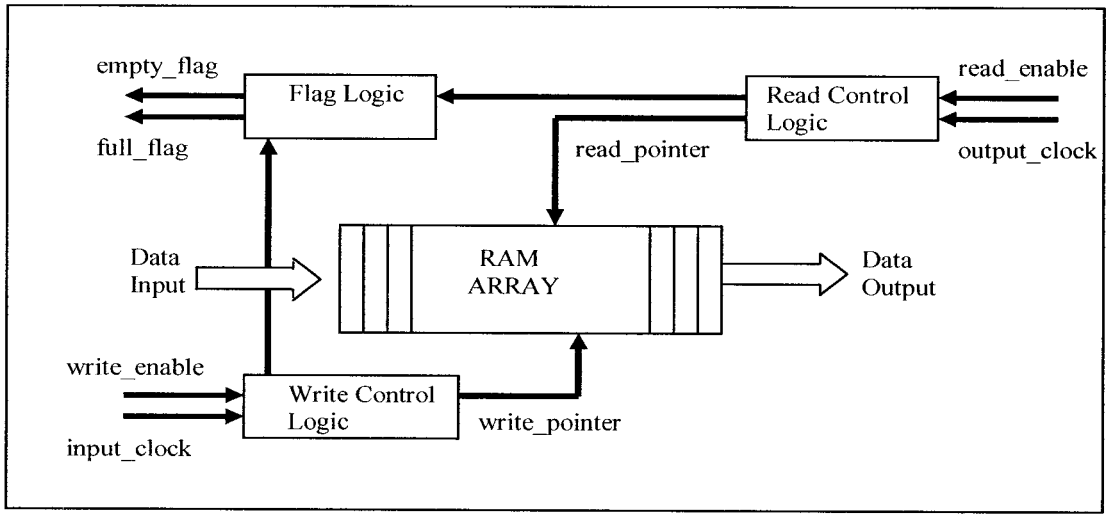


Figure 3.5: RAM-based FIFO [39]

3.3.1 The Implementation of a FIFO

In this section, a diagram of the FIFO is described. This design is divided into two parts: one is the FIFO logic, the other is the FIFO memory. For the FIFO logic part [26], the

input and output signals are defined in Table 3.1 and the Verilog code for logic control of the FIFO is listed in appendix A-1.

Table 3.1: Signal Type Definition for a FIFO

Signal Name	Signal Type	Signal Type Definition
clk_in	input	Write clock signal
clk_out	input	Read clock signal
*reset_*	input	Low asserted reset signal
data_in	input	Data into FIFO
fifo_wr_en	input	Enable write data into FIFO
fifo_rd_en	Input	Enable read data from FIFO
data_out	output	Data output from FIFO
full_flag	output	Flag of FIFO full
empty_flag	output	Flag of FIFO empty

3.3.1.1 The FIFO Logic

A FIFO is used as a "First-In First-Out" memory buffer between two synchronous logic blocks with simultaneous write and read data to or from the FIFO. These data accesses are independent of each other when the FIFO is implemented using a dual-ported memory. Each port has its own corresponding pointer that points to a location in memory. When a FIFO is reset, both the write and read pointers point to the first memory location. Each write operation causes the write pointer to point to the next address (new location) in memory and the counter is also increased. Similarly, each read operation causes the

read pointer to point to the next address (new location) in memory and the counter is decremented by one. The FIFO write and read operations loop in a circular fashion from the last memory location to the first memory location without resetting the read and write pointers (Figure 3.6).

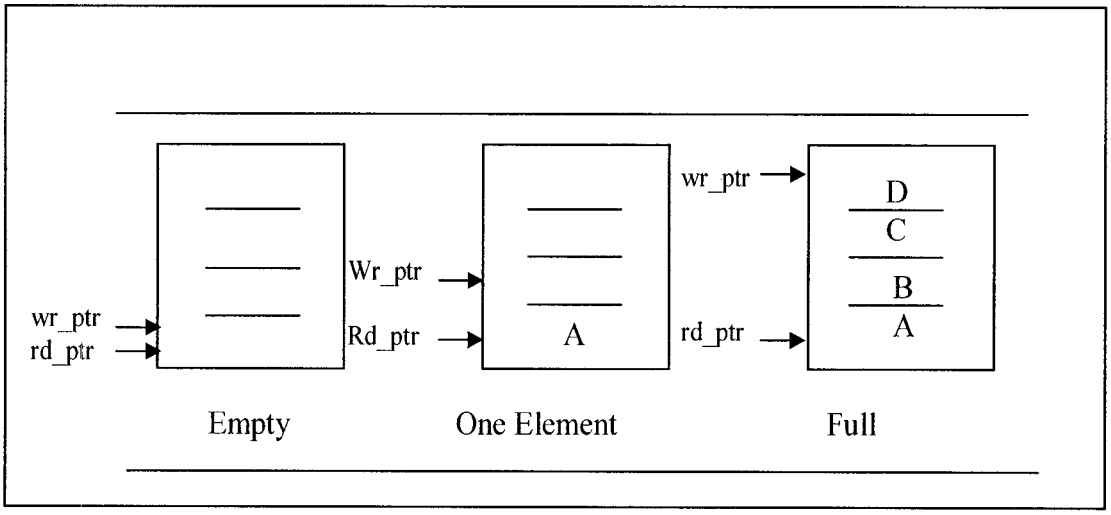


Figure 3.6: The Operation of Read and Write Pointer in Memory

There are two status flags in the FIFO logic unit. Each flag indicates whether the FIFO status is empty or full. The output of both flags depends on the value of the counters. If the value of the counter is zero, an empty flag is asserted. If the value of counter is equal to the depth of the FIFO minus one (D-1, initialize location is one), a full flag is asserted. Otherwise, the FIFO contains data that can be accessed when the *fifo_rd_en* or the *fifo_wr_en* signal is enabled (Figure 3.7).

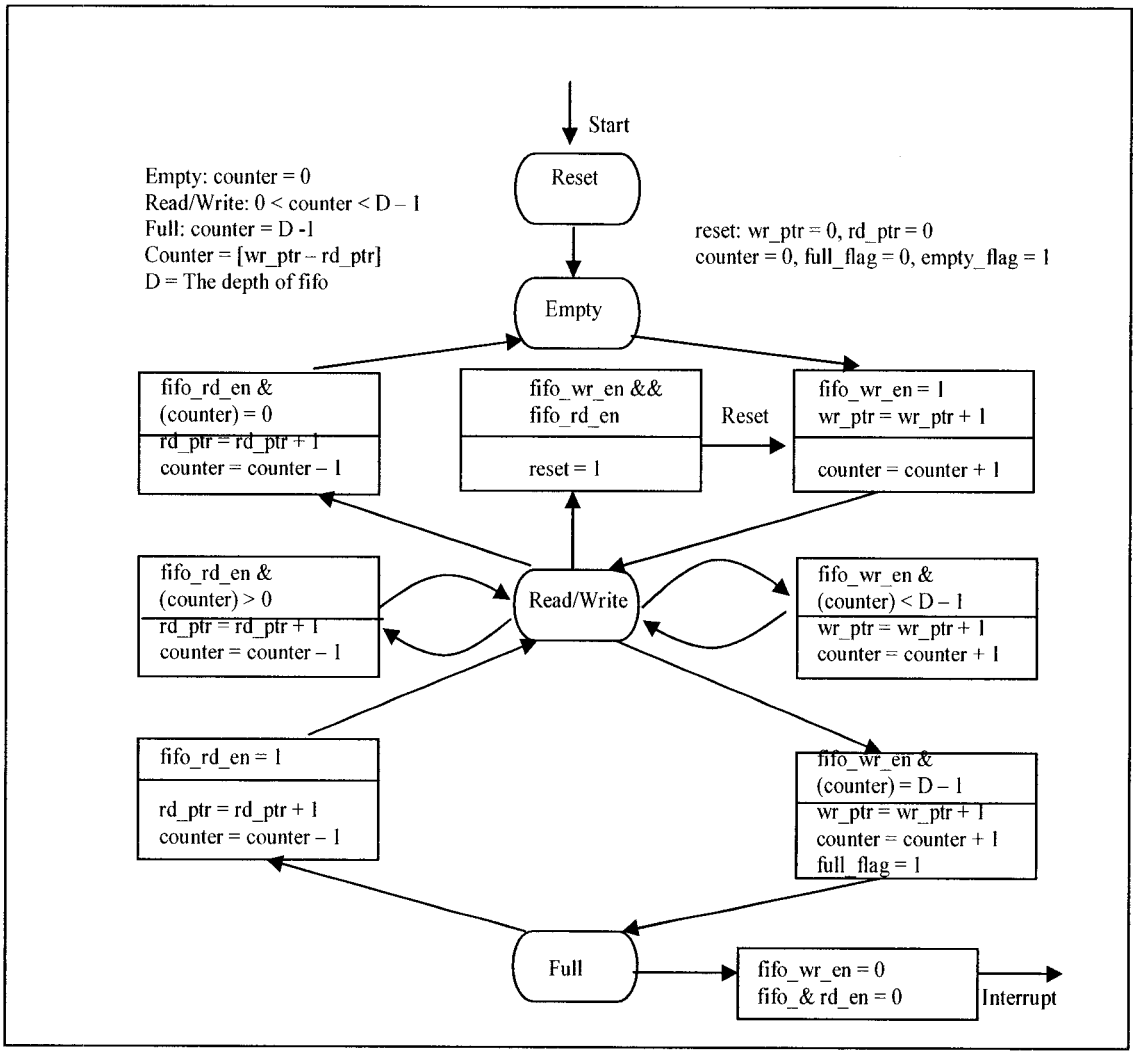


Figure 3.7: Finite State Machine of FIFO (FSM)

3.3.1.2 The FIFO Memory

The FIFO memory is a synchronous RAM (SRAM) consisting of dual-port cells [26]. Its schematic diagram is shown in Figure 3.8. Each dual-port cell is composed of a standard six-transistor memory cell, with two additional transistors to implement the dual port

function. The BIT LINE P1 writing operation and The BIT LINE P2 reading operation are not correlated at any time.

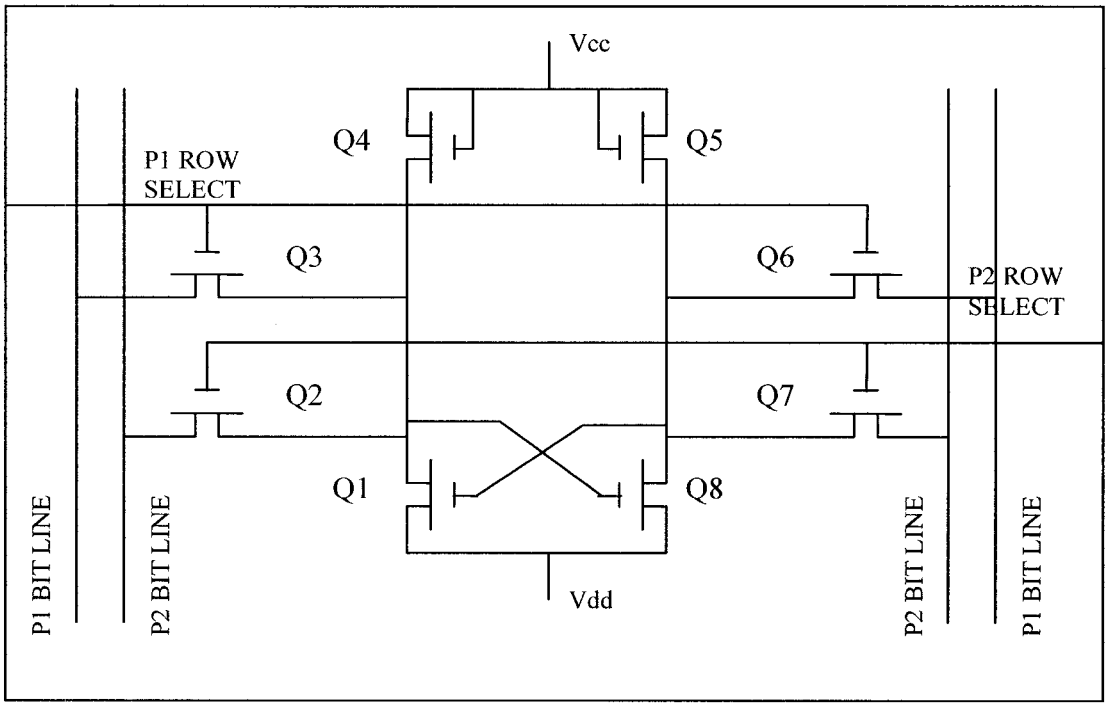


Figure 3.8: Dual-Port Cell for the FIFO Memory [26]

CHAPTER 4
MULTIPLE-HIGH-FREQUENCY SYNCHRONOUS CLOCK GENERATOR

4.1 All Digital Delay Locked Loop and Mirror Delay Approach

The Multiple-High-Frequency Synchronous Clock Generator (SCG) may work as a standard clock for the MEPG-2/ATM/SONET interface's to data storage or transmission and receiving functional units. The SCG generates multiple-high-frequency clocks, the generated frequencies are 38.88 MHz, 77.76 MHz and 155.52 MHz. All the functions are implemented using purely digital components. The design is based on the All-Digital Delay Locked Loop (ADDLL) function and Mirror Delay Approach (MDA). The ADDLL function is used to adjust the delay of a control phase locked loop; again using completely digital circuits. The resulting design is insensitive to variations in temperature, voltage supply and process techniques. There is also a reduction in frequency jitter and clock skew due to the clock distribution between chips and timing recovery in serial data communication. MDA uses the principle of a mirror to control multiple-stage proportional delays. We present in this project an innovative application of both methods, ADDLL and MDA, to generate a multiple-high-frequency synchronous clock. The circuit is implemented with a CMOS 0.35 μm process.

To obtain different data transport rates at the input and output interfaces, a fast clock generator (internal clock) is necessary. This clock must be closely synchronized with the external standard clock. The most popular solution to this synchronization problem is to use a Phase Locked Loop (PLL) [17, 20]. At first, analog PLL technology was used, but

with the improved performance of digital devices, more designers are using digital PLLs. In many existing Digital Phase-Locked Loops (DPLL) [31, 21], the only digital component is a Phase Detector (PD), while the other components, such as the Voltage Controlled Oscillator (VCO) and Loop Filter (LF), are analog components with all the problems inherent to analog hardware. The ADDLL is built exclusively from digital circuits. Therefore, it does not contain any passive components such as resistors and capacitors. The Loop Filter and Charge Pump are replaced by either a Universal Shift Register (USR) or a Counter and a Delay Line (DL) [22, 42].

Since an ADDLL can be used in clock synchronization as a DPLL, previous implementations have shown that the slave clock must be closely synchronized with the master clock using the ADDLL technique [14, 27]. A seven-bit counter is used instead of an analog loop filter to select a delay line via the decoder. This type of the delay line is fed to the output buffer. Other authors [6] describe a register-controlled DLL circuit (RDLL), where the internal clock is synchronized with an external clock. A Universal Shift Register (USR) is used; this register inserts an optimum delay time between the external clock input buffer and the internal clock output buffer. The internal clock changes simultaneously with the next rising edge of the external clock. In summary, the RDLL is only applicable for the synchronization of clock, and is locked to a detected clock frequency with an external input clock frequency. Both clocks operate at the same frequency. Our implementation is the first application using an ADDLL function and MDA to generate multiple-high-frequency synchronous.

4.2 Basic Idea of the Multiple-High-Frequency SCG

This design is divided into two sections as shown in Figure 4.1: one is the ADDLL, and the other is the Frequency Multiplier (FM). The Delay Line (DL) circuit is used only for aligning the phase difference between a detected signal and a reference signal in the ADDLL section, which is equivalent to aligning the output signal of each stage of the FM. Since the circuit architecture of the DL is similar in these two sections, their delay characteristics on the same chip are consistent and in a mirror proportional relationship.

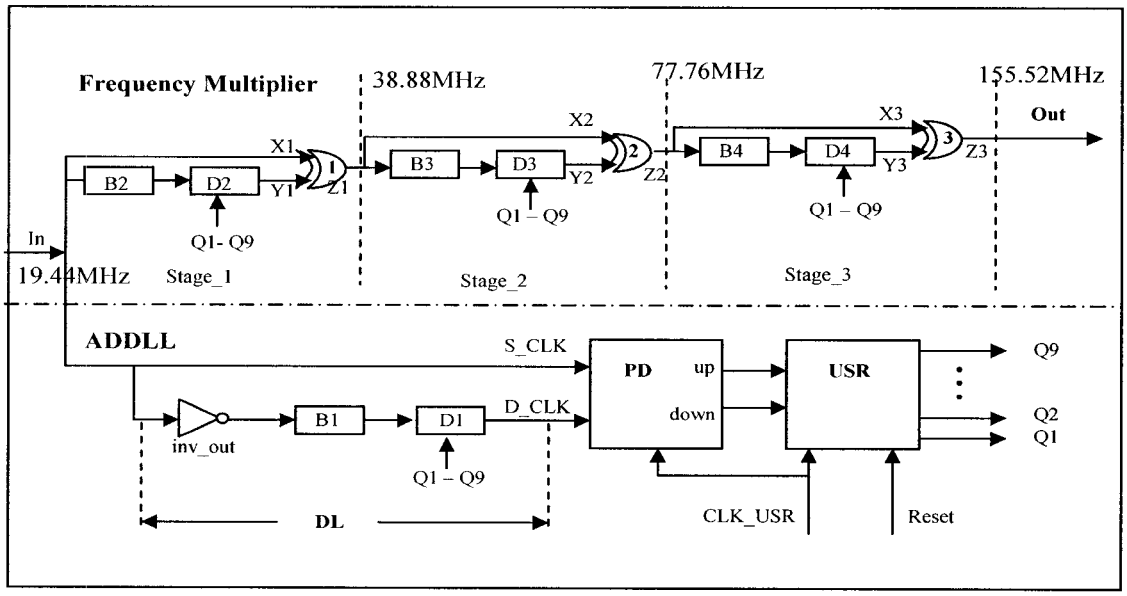


Figure 4.1: Schematic of ADDLL and MDA for Multiple-High-Frequency SCG

There are four sets of delay lines labelled 1, 2, 3 and 4. In Figure 4.1, "Bx" represents a fixed delay line and "Dx" represents a variable delay line (where, x = 1, 2, 3 and 4). The delay time is represented individually by "TBx" and "TDx" (x = 1, 2, 3 and 4). The sum of these delays, for each stage is Tx, where Tx = TBx + TDx.

The FM is used to generate a multiple-high-frequency clock signals. Each stage is composed of a XOR gate, a fixed delay line "Bx" and a variable delay line "Dx". There

are two inputs in each XOR gate, they should have different input times. For example, when one input is high (X1, X2, and X3), the other input (Y1, Y2, and Y3) goes high after a quarter cycle of X1, X2 and X3, as demonstrated in Figure 4.2. As for the ADDLL section, its function is to lock a clock phase (*D_CLK*) with a reference clock phase (*S_CLK*) by control of the variable delay line "D1". Because the variable delay lines D2, D3 and D4 are connected with the variable delay line D1 by signals $Q1 \sim Q9$, the variable delay times for D2, D3 and D4 are controlled by the adjustment of D1 (Figure 4.1). The output (Z1, Z2, and Z3) of each stage of the FM are locked with the *S_CLK* signal when the *D_CLK* is locked with the *S_CLK* signal.

If the *D_CLK* signal is locked with the *S_CLK* signal in the ADDLL section, then it is equivalent to being locked to the output of each stage in the FM. This is also called a Pseudo-ADDLL circuit because it is a replica circuit, which does not directly generate multiple-high-frequency signals. In other words, this circuit acts like a mirror to symmetrically copy the delay time of each stage in the FM. Thus, this approach is called a Mirror Delay Approach.

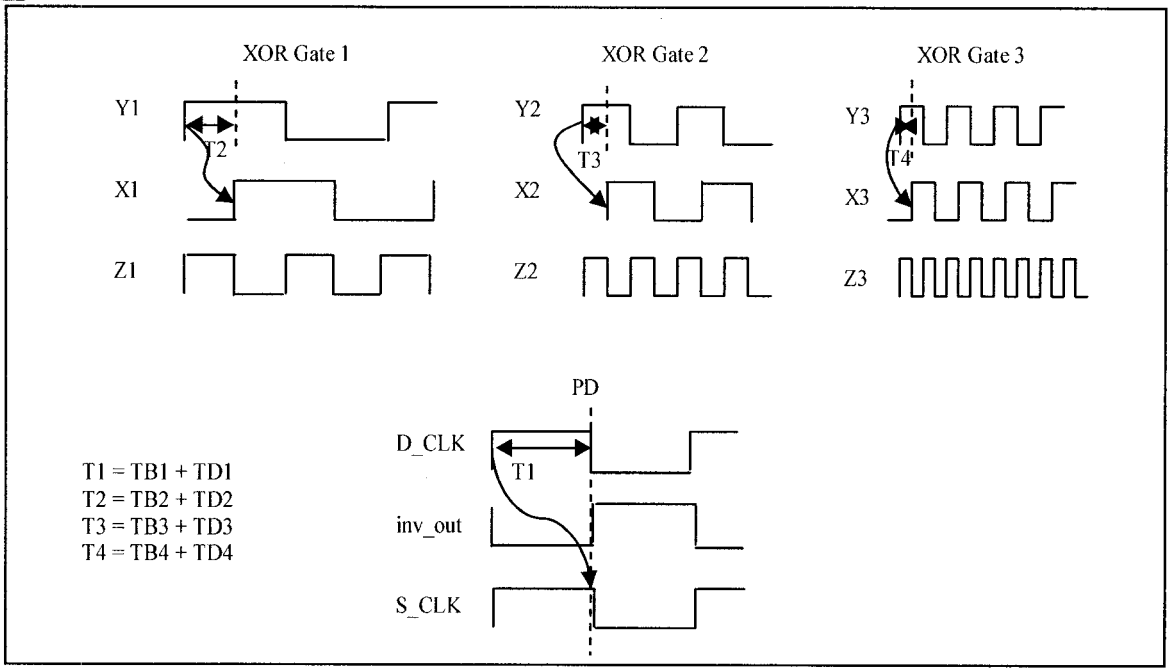


Figure 4.2: Input and Output Waveforms of Each Gate and Phase Detector

As illustrated in Figure 4.1, the DL of each stage in the FM section and the DL in the ADDLL section are divided into two parts. One part generates a fixed delay time TBx (x = 1, 2, 3 and 4) and accounts for about 80 percent of the sum delay time Tx. The other part generates a variable delay time TDx (x = 1, 2, 3 and 4), which accounts for the remaining 20 percent of the sum delay time Tx. The TDx variable delay time may be adjusted by the Universal Shift Register (USR) because the D1, D2, D3 and D4 variable delay blocks are connected together by identical *Q1* to *Q9* signals in the USR; the delay time variation of TD1, TD2, TD3 and TD4 also have the same ratio. The output of each stage on the FM (Z1, Z2, and Z3) and the *D_CLK* signal in the ADDLL section are synchronous variations with a 2^n times relationship. Obviously, this circuit only detects the phase state of the *D_CLK* signal and locks it with the *S_CLK* signal, therefore, locking the output of each stage in the FM section.

The delay time of fixed delay TBx and the delay time of variable delay TDx in each delay block have a proportional, 2^n times relationship (n = 3, 2, 1, 0) as shown in table 4.1. In this table, P_1, P_2 and P_3 represent the input clock period of each stage in the FM section (X1, X2, X3 signals, in Figure 4.1). For example, P_1 = 48ns, P_2 = 24ns and P_3 = 12ns.

Table 4.1: 2^n Relationship for Each Delay time TBx and TDx (x= 1, 2, 3 and 4)

Fixed Delay Time	**TB1 $Cx2^3$**	**TB2 $Cx2^2$**	**TB3 $Cx2^1$**	**TB4 $Cx2^1$**
(ns)	~80%(P_1/2)	~80%(P_1/4)	~80%(P_2/4)	~80%(P_3/4)
C= 2.557	20.57	10.28	5.14	2.557
Variable Delay Time	**TD1 $Cx2^3$**	**TD2 $Cx2^2$**	**TD3 $Cx2^1$**	**TD4 $Cx2^1$**
(1~9) (ns)				
C=0.066	0.528	0.264	0.132	0.066

In our design, T2, T3, and T4 represent the input delay time of each XOR gate in the FM section, T1 represents the input delay time of DL in the ADDLL section. In order to synchronize the rising edge of the *D_CLK* signal with the rising edge of the *S_CLK* signal, T1 (T1 = TB1 + TD1), one input delay time, must be a half period of the *S_CLK* signal. Similarly, the input delay time of D2, D3, and D4 are also adjusted using a 2^n times relationship and the delay times T2, T3, and T4 will be quarter period of the *X1, X2,* and *X3* signals. Therefore, signals *Y1, Y2,* and *Y3* each have delay times representing T2, T3, and T4. When the variable delay line D1 is locked such that the *D_CLK* signal is closely synchronized with the *S_CLK* signal, then *Z1, Z2,* and *Z3* will be operating at the exact high frequencies required and will be closely synchronized with the *S_CLK* signal.

The input and output waveforms of each XOR gate in the FM section and the DL in the ADDLL section are shown in Figure 4.2.

4.3 SCG Architecture

The SCG is composed of four main functional blocks:

- Phase Detector
- Universal Shift Register
- Delay Line
- Frequency Multiplier

4.3.1 The Phase Detector

A block diagram of the Phase Detector (PD) is shown in Figure 4.3. The output of the PD depends on the phase of the input signals. This type of phase detector is also called a sequential phase detector. It compares the rising edge of the detected clock (*d_clk*) and the rising edge of the standard clock (*s_clk*).

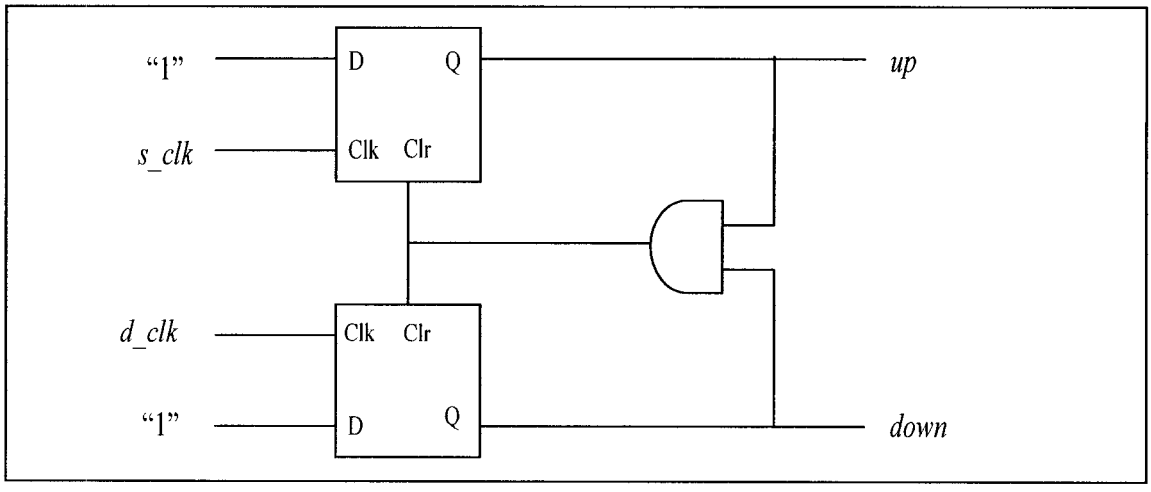


Figure 4.3: Block Diagram of the Phase Detector

If the rising edge of *d_clk* leads the rising edge of *s_clk* (Figure 4.4-a), the "*down*" output of the phase detector goes low while the "*up*" output remains high. Also, if the rising edge of *d_clk* delays the rising edge of *s_clk* (Figure 4.4-b), the "*up*" output of the phase detector goes low while the "*down*" output remains high. Finally, if the rising edge of *d_clk* and the rising of *s_clk* are synchronous (Figure 4.4-c), the output of "*up*" and "*down*" both remain low.

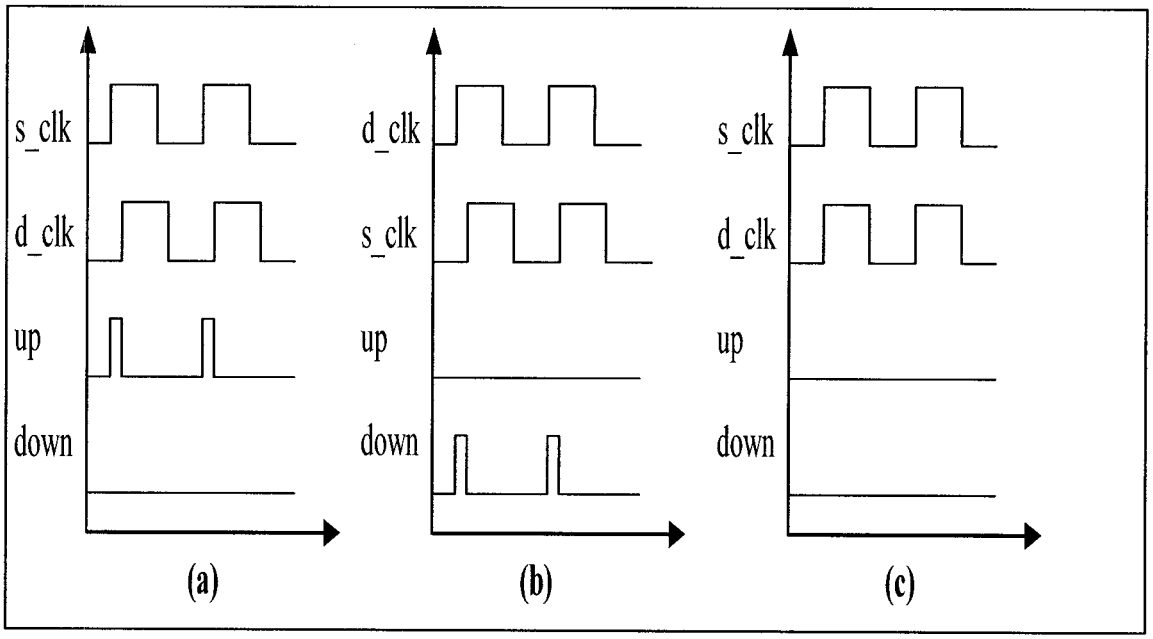


Figure 4.4: Input and Output Waveforms of the Phase Detector

4.3.2 The Universal Shift Register

The Universal Shift Register (USR) performs the following functions: Hold value, Shift left, Shift right, and Load value [5].

This USR is used as a parallel-in, parallel-out shift register. The following table shows the states for the 9-bit universal shift register.

Table 4.2: 9-bit Universal Shift Register

Function	Input S_1S_0(down/up)	Initial State Q1 Q2 Q3 Q4 **Q5** Q6 Q7 Q8 Q9
Hold	0 0	Next State Q1 Q2 Q3 Q4 **Q5** Q6 Q7 Q8 Q9
Shift left	1 0	Q2 Q3 Q4 Q5 **Q6** Q7 Q8 Q9 Q1
Shift right	0 1	Q9 Q1 Q2 Q3 **Q4** Q5 Q6 Q7 Q8
Load	Reset = 1	0 0 0 0 **1** 0 0 0 0

4.3.3 The Delay Line

The Delay Line (DL) is composed of two parts: fixed delay and variable delay lines. The fixed delay line is made of several pairs of inverters. The variable delay line is made of nine stages using NAND gate blocks. The variable delay line is used to insert an optimum delay time, and make the *D_CLK* signal change simultaneously with the next rising edge of the *S_CLK* signal, which is controlled by the universal shift register. A simplified schematic of variable delay line is shown in Figure 4.5.

Only one of the Q1 to Q9 inputs is loaded with a "1"; the remainder of the inputs are loaded with "0"s. The gate loaded with a "1" will be used as an output path. The different output paths have different delay times for the output of the DL in the ADDLL section.

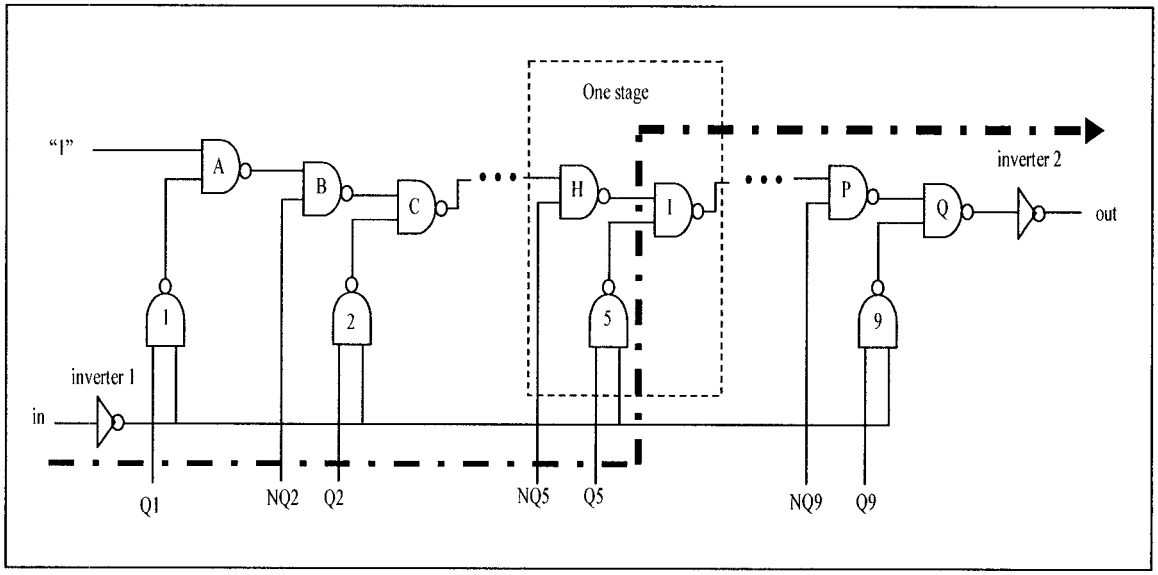


Figure 4.5: Schematic of Variable Delay Line

Initially, the DL input is loaded with the pattern "0, 0, 0, 0, 1, 0, 0, 0, 0". With NAND gate 5 set to "1", the input signal is proceeds from inverter1 to NAND gate 5, then through the I NAND gate, then passing through the remaining stages to the P and Q NAND gates and finally through inverter 2 to the output (Figure 4.5). In the next step, the delay time "TD1" can be increased or decreased depending on the output of the PD. If the *D_CLK* signal is delayed with the *S_CLK* signal as detected by PD, then the NAND gate 6 is set to a "1", the signal path is through inverter1, NAND gate 6, NAND gate K... NAND gate P, NAND gate Q and finally through inverter 2. Thus the delay time "TDx" of the output signal is decreased. If the *D_CLK* signal leads the *S_CLK* signal as detected by PD, then NAND gate 4 is set to a "1", and the signal path is through inverter1, NAND gate 4, NAND gate G, ... NAND gate P, NAND gate Q and inverter 2. The delay time "TD1" of the output signal is increased. If the *D_CLK* signal is locked with the *S_CLK*

signal, then the output of variable delay line D1 does not changed. The signals Q1 to Q9 in the variable delay line D1 are controlled by the PD.

4.3.4 The Frequency Multiplier

The Frequency Multiplier (FM) is responsible for generating multiple-high-frequency signals at 38.88 MHz, 77.76 MHz and 155.52 MHz from the input signal 19.44 MHz. A waveform of one stage Multiplier Frequency is shown in Figure 4.6. The variable delay line block "Dx" (x = 2, 3 and 4) has the same architecture as the variable delay line D1 block in the ADDLL section. The propagation delay at point A is controlled by the USR in the ADDLL section (Figure 4.6). To synchronize the outputs of the multiple-high-frequency signals in the FM section with the rising edge of the input signal (In), one chain of the circuit in Figure 4.6 is needed. For example, if the FM needs to generate an output signal which is eight times the frequency (2^3) of the input clock, three stages of the circuit in Figure 4.6 are chained.

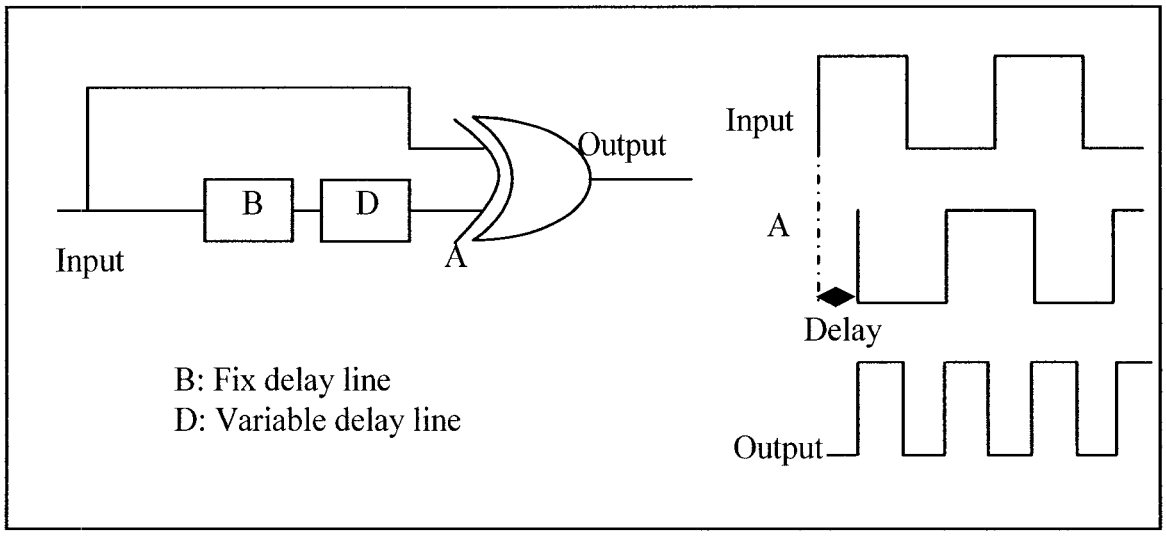


Figure 4.6: Waveform of One Stage Frequency Multiplier

4.4 Implementation of the Synchronous Clock Generator

Special attention was paid to the following issues were noticed during the implementation of the SCG using a 0.35-μm CMOS process library [37] from TSMC:

- Clock of the USR and glitching noise
- Timing constraints on the USR cell
- Driving large capacitive loads
- Design inverter layout cell for the fixed delay line

4.4.1 Clock of the USR and Glitching Noise

As previously discussed, the function of the USR is to shift the current state to the next state according to the output signal of the PD in the ADDLL section. As a result, a clock (*CLK_USR_1*) must control the shift operation of the USR (Figure 4.7).

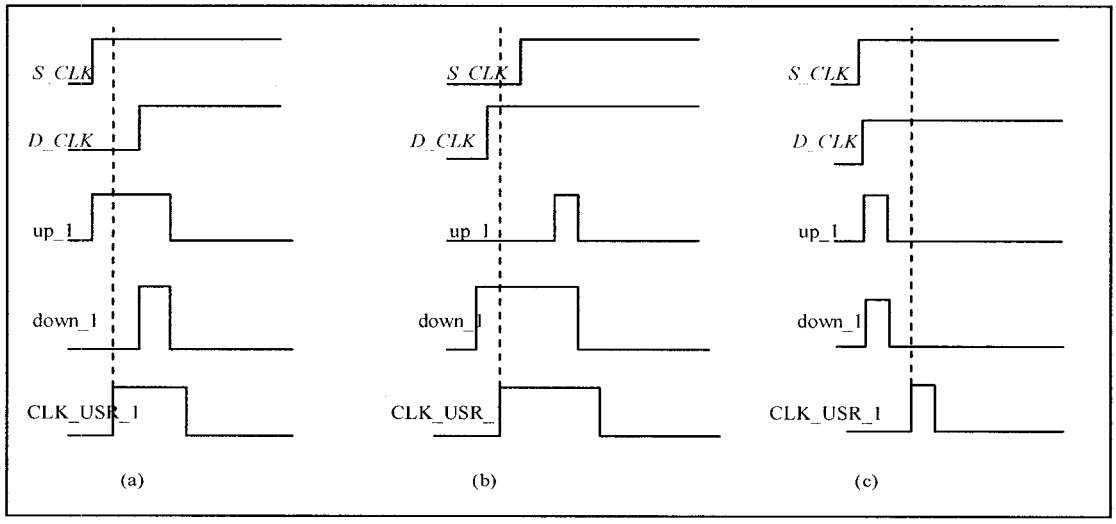


Figure 4.7: Waveforms without flip-flops in the USR: (a) *D_CLK* lead with *S_CLK*, (b) *D_CLK* delay with *S_CLK*, (c) *D_CLK* synchronized with *S_CLK*

During the synchronization process the rising edge of *CLK_USR_1* must be triggered between the rising edge of the *S_CLK* signal and the rising edge of the *D_CLK* signal. The rising edge of *CLK_USR_1* must be triggered on "*up*"= 0 and "*down*" = 0, when *D_CLK* is synchronized with *S_CLK*. The waveforms without the flip-flop delays are as shown in Figure 4.7-(a) and 4.7-(b).

By adding an OR gate and buffers (several pairs of inverters), an output signal is obtained which acts as a clock with a suitable delay (Figure 4.8). This suitable delay prevents the output from becoming unstable when the rising edges of *D_CLK* and *S_CLK* both occur simultaneously. Its waveform is shown just in Figure 4.7-(c).

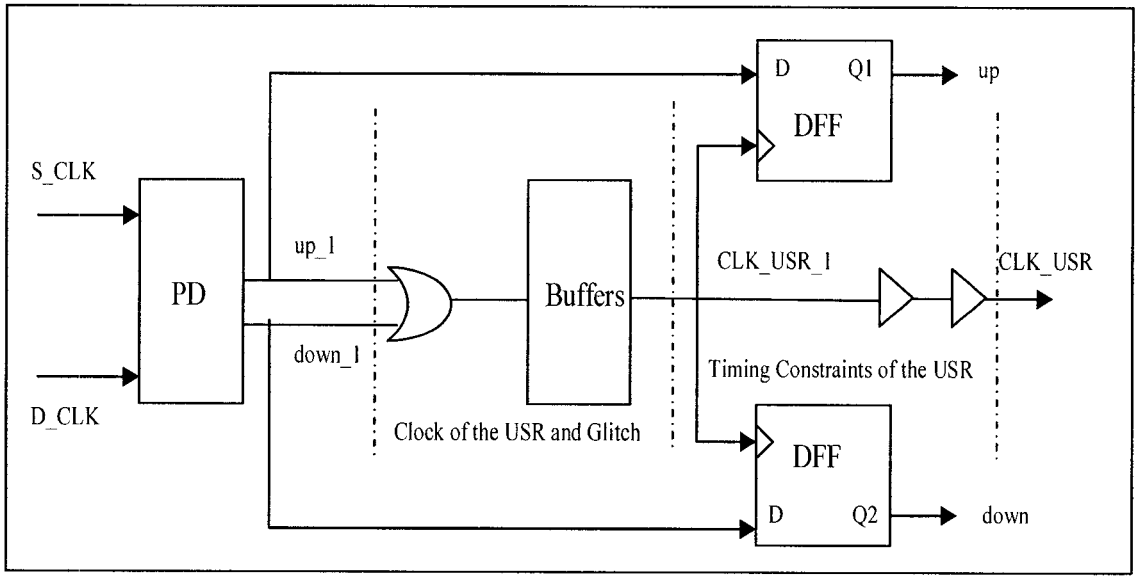


Figure 4.8: Schematic of the PD with DFF

4.4.2 Timing Constraints of the USR Cell

The USR is an edge sensitive cell with associated setup time and hold time requirements. The timing constraints of the USR cell are illustrated in Figure 4.9. The excitation inputs

S0 (Q1 / up) or *S1 (Q2 / down)* of the USR should remain stable immediately before or after the clock transition. As defined for an edge-triggered flip-flop, the period before the clock transition for the USR is defined to be the setup time (t_{su}); the period after the clock transition for the USR is defined to be the hold time (t_h). The minimum width of the synchronizing clock pulse is defined to be the pulse width (t_w). The time interval from the triggering edge of the clock to the stabilization of the new state (cell output) is defined to be the propagation delay (t_p). In general, if these specified constraints for the USR cell are violated, the cell's behaviour is not guaranteed.

For this implementation, the *up_1* and *down_1* signals (Figure 4.9) are not stable enough for the USR. During a clock transition, these signals change too rapidly for the USR. In order to solve this problem, two edge-triggered flip-flops (DFF) and two buffers are added (Figure 4.8). The value of *up_1* and *down_1* signals are latched into DFFs when the rising edge of *CLK_USR_1* occurs, the output *Q1* and *Q2* of DFFs will keep the value of *up_1* and *down_1* until the next rising edge of *CLK_USR_1*. Thus, the instantaneous state value of the *up_1* and *down_1* signals before or after the clock (*CLK_USR_1*) transition will be transformed into stable state values at the *Q1* and *Q2* outputs. These input signals of USR cell, *S0 (Q1/up)* and *S1 (Q2/down)*, meet the setup time (t_{su}) and hold time (t_h) requirements of the USR cell.

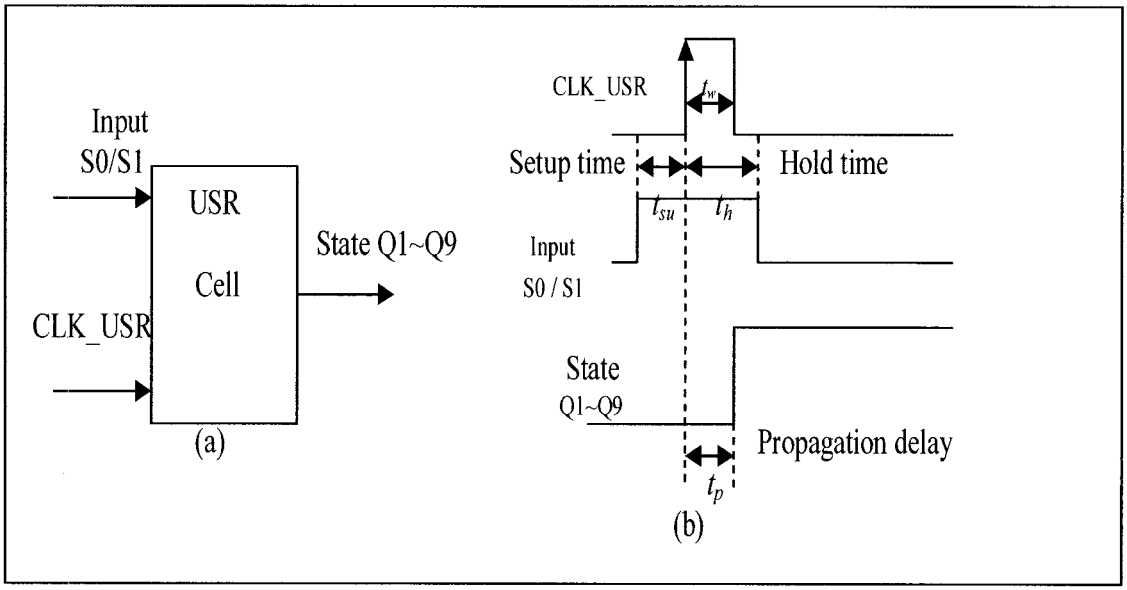


Figure 4.9: Timing Constraints of the USR Cell

4.4.3 Driving Large Capacitive Loads

The problem associated with driving comparatively large capacitive loads arises when signal must be propagated from one block to another block on-chip. Since an output to drive next block has a definite limit on the amount of current it can supply (or sink), it is not surprising that there is a definite limit on the number of parallel switches that can be driven by a signal switch output. This limit (number) is called the fan-out of the block. The overall performance of a system may be seriously degraded if it contains a large fan-out, where one circuit within the system is required to drive a large capacitive load. This situation often occurs in the case of control drivers that are required to drive a large number of inputs to memory cells or logic blocks.

The solution is to use a pair of cascaded inverters or buffers, where each one is larger than the preceding stage by a width factor *f*, as shown in Figure 4.10. Notice that if a

large factor f is chosen, only a few stages may be needed to achieve a large enough current capacity, but each stage will have a long delay. If a small factor f is chosen, the delay of each stage is shorter, but more stages are required.

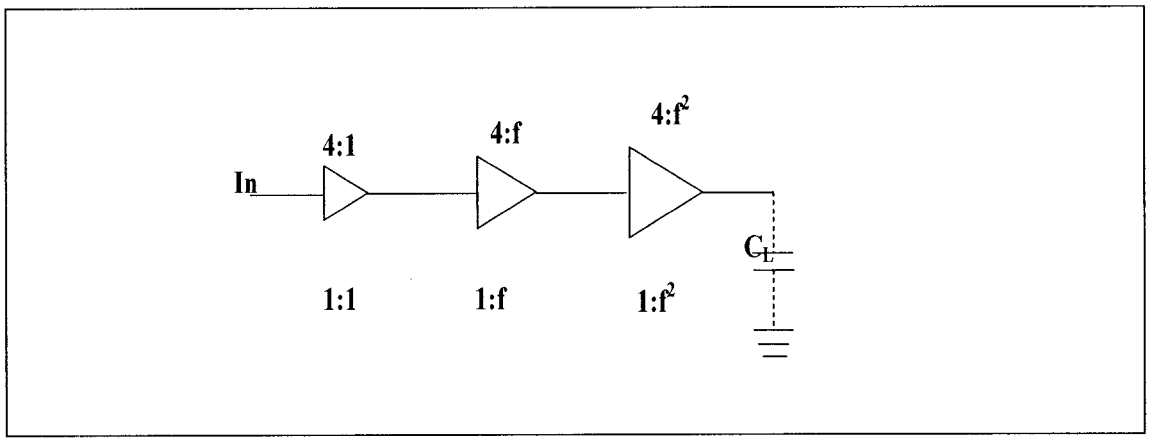


Figure 4.10: Driving Large Capacitive

For this implementation, the buffer chain is made of wbuf_1, wbuf_2 and wbuf_4 from the TSMC CMOS 0.35-μm [35].

4.4.4 Design Inverter Layout Cell for the Fixed Delay Line

The fixed delay line is composed of a series of inverters pairs. The type of inverter available in the TSMC CMOS 0.35-μm process, have a delay time of a hundred pico-second (10^{-12}). However, the delay time of fixed delay line on our design is in the order of nano-seconds (10^{-9}). Several inverters are required to generate sufficient delay time, but using many inverters will make the schematic level simulation very difficult. Therefore, a better choice is to use custom design for a special inverter layout cell. The propagation delay of the inverter is increased by expanding the width of the layout area,

reducing the number of inverters needed in the fixed delay line, and simplifying the schematic level simulation [34].

4.5 Chip Characteristics and Design Flow

Figure 4.11 shows the SCG IC layout using the 0.35-μm CMOS process library provided by TSMC. The additional poly1 in the empty areas of the layout occupies 14 percent of the chip area. Table 4.3 summarizes the chip's features (ICDPMLIU).

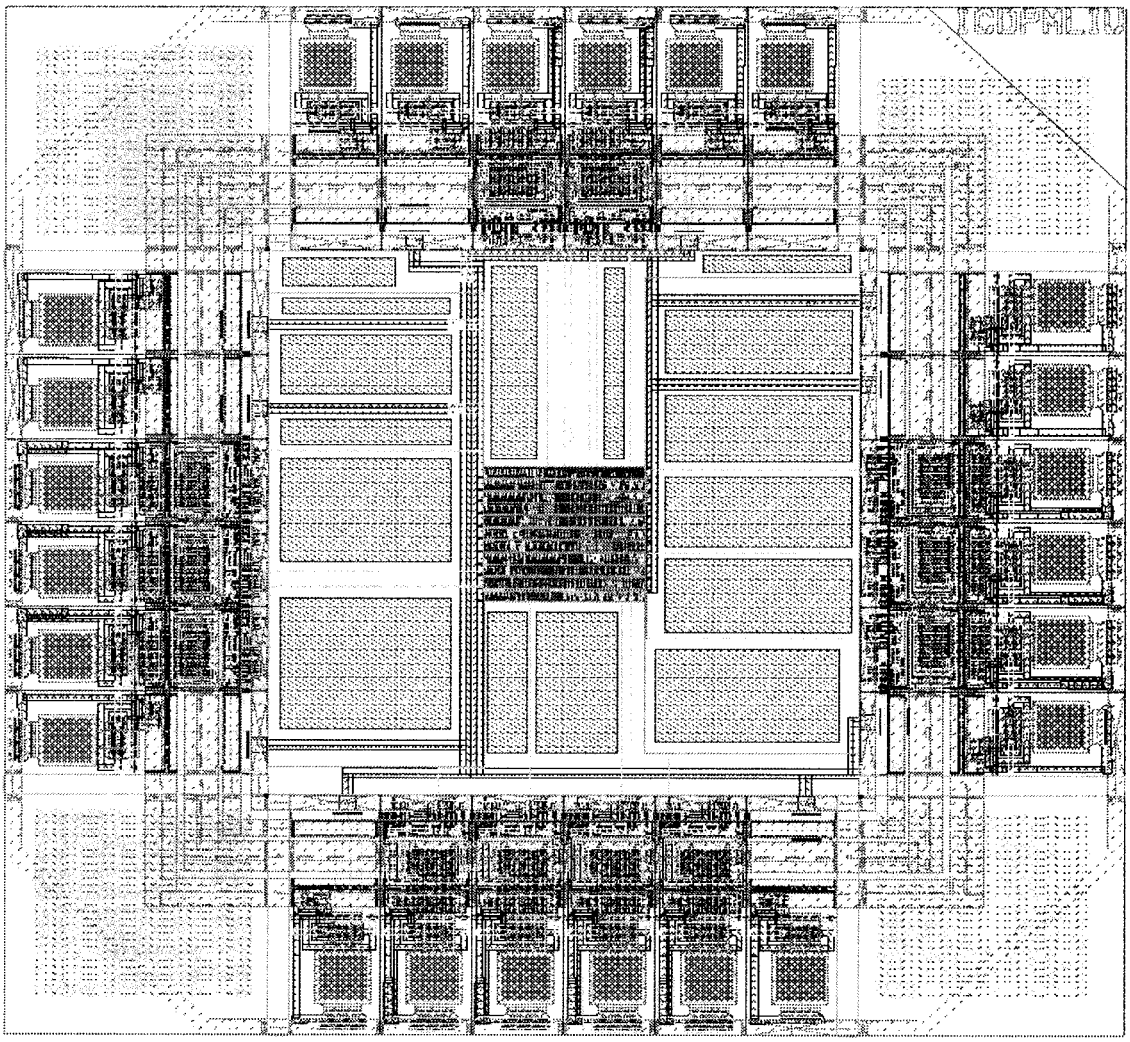


Figure 4.11: Layout of Chip ICDPLIU

Table 4.3: Summary of the Chip's Features (ICDPMLIU)

Process	CMOS 0.35-μm, TSMC
Chip Size	1982.5 x 1982.5 μm
Cell size	287.4 x 261. 3 μm
I/O Pads	24
Input Frequency	19.44 MHz
Output Frequency	38.88 MHz, 77.76 MHz and 155.51 MHz
Supply Voltage	3.3 V

The design sources are VHDL code (provided by Synopsys) for the USR cell and schematic (provided by Cadence) design for the PD, DL, and FM cells. For layout design, both AutoLayout and manual design methods are used. The following design software packages were used in the design: USR cell, VHDLDBX for functional simulation and DESDIGN_ANALYZER for synthesis. The SPECTRE simulation tool is used to simulate the PD, DL, FM cells and the entire SCG system at the schematic level [36]. This design flow is shown in Figure 4.12.

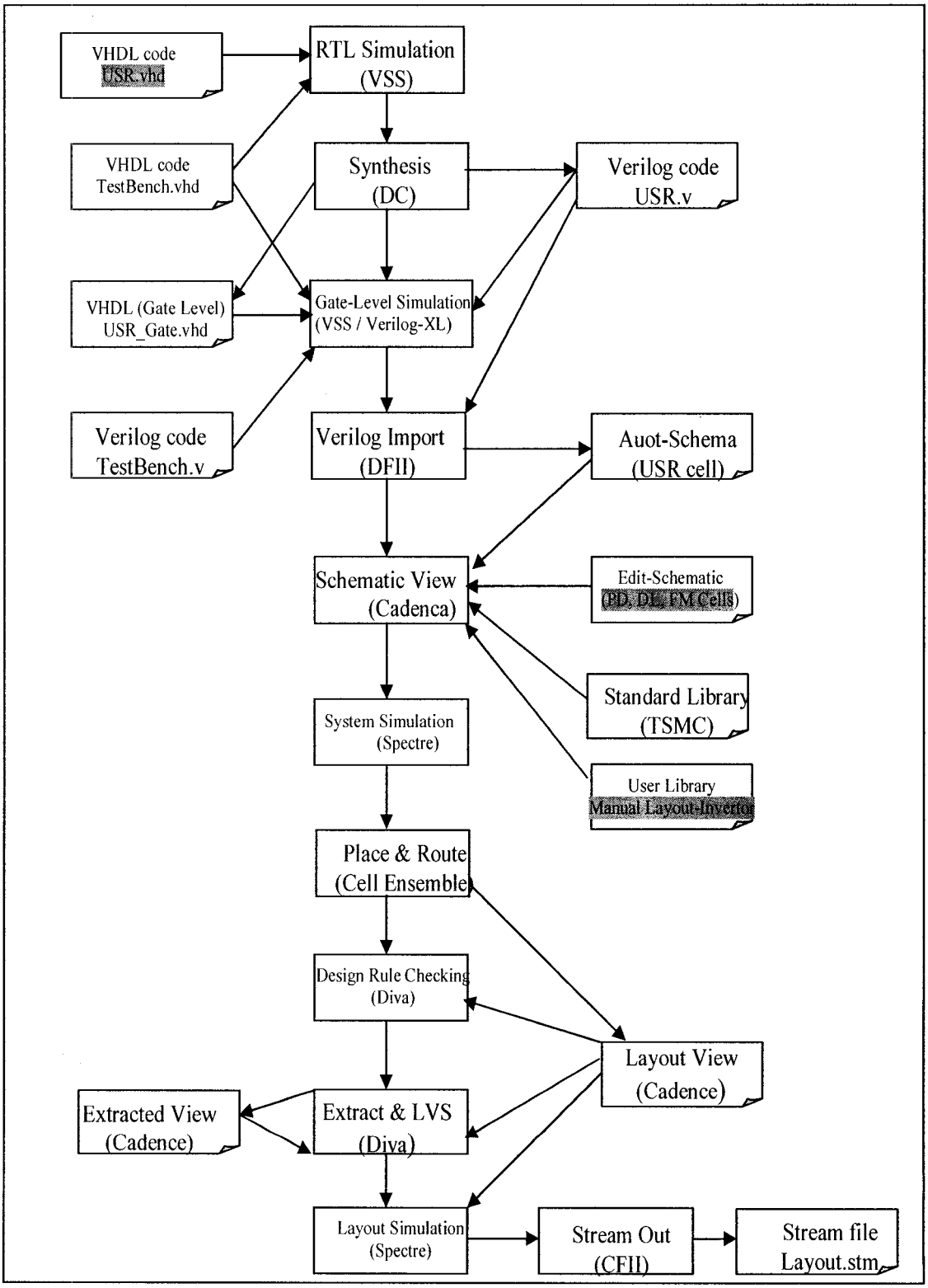


Figure 4.12: Design Flow for the Multiple-High Frequency SCG [9]

4.6 ADDLL Function for Timing Recovery

As discussed in previous sections, the ADDLL is used as a new method to generate multiple-frequency transmission clock, which is phase locked with an external standard clock. This method guarantees that the transmission data is synchronized with its clock signal. In this section, the ADDLL function is used for timing recovery. In Figure 4.13, at the reception side, the incoming high-speed SONET bitstream signal and the internal clock are fed to the Alexander phase detector [33].

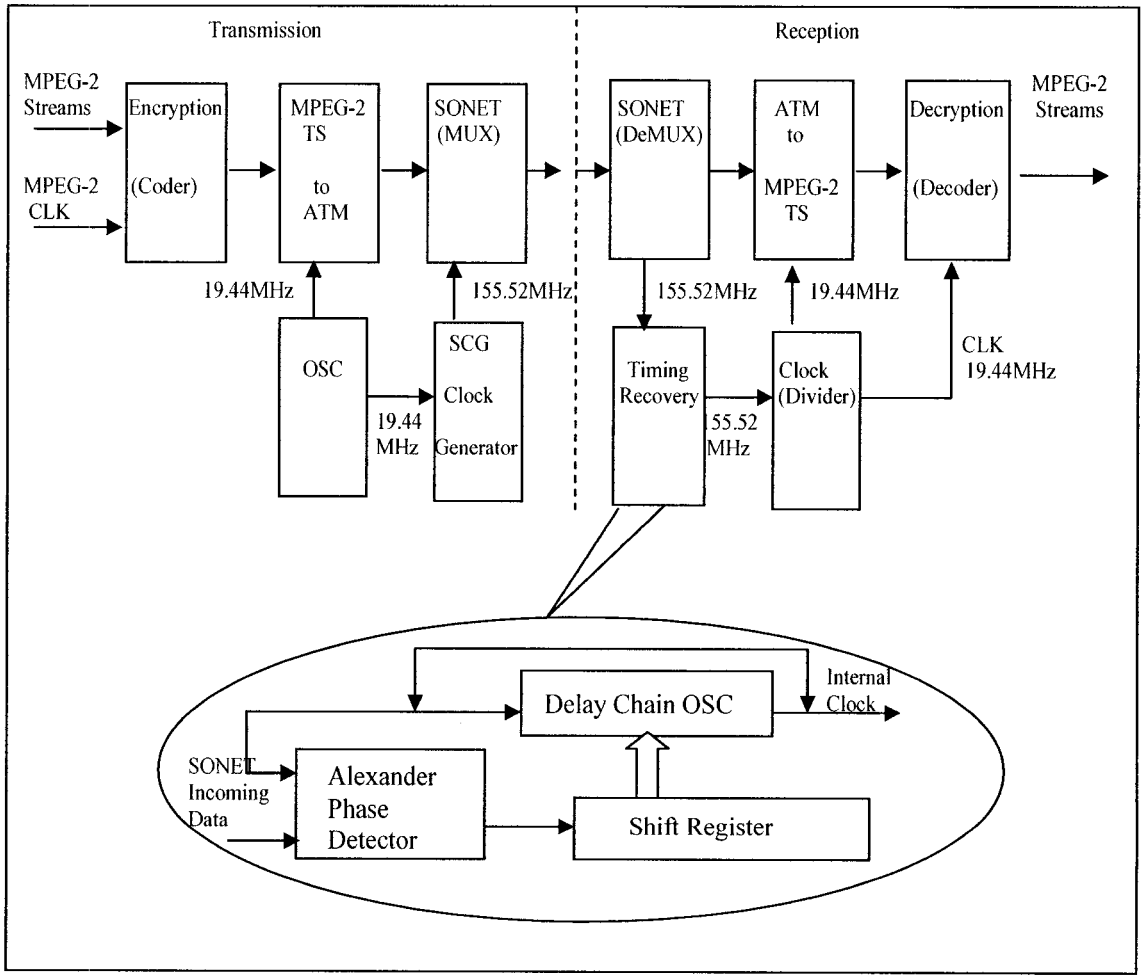


Figure 4.13: Application of ADDLL Function for Timing Recovery

The bitstream signal is compared with the internal clock. If these signals are out of phase, the phase detector outputs an up/down flag to the shift register. Adjustments continue to delay chain oscillator until the internal clock and bitstream signal are in phase. The delay chain's control is the same as the SCG control. The main difference with the SCG is the Alexander phase detector.

The data sampling operation is shown in Figure 4.14. The "eye pattern" of the incoming data is sampled at three points in time: A, T, and B. When the clock and data signals are synchronized (locked), the following is true:

- A is the centre of the data bit
- T is in the vicinity of the next transition
- B is in the centre of the bit following the transition

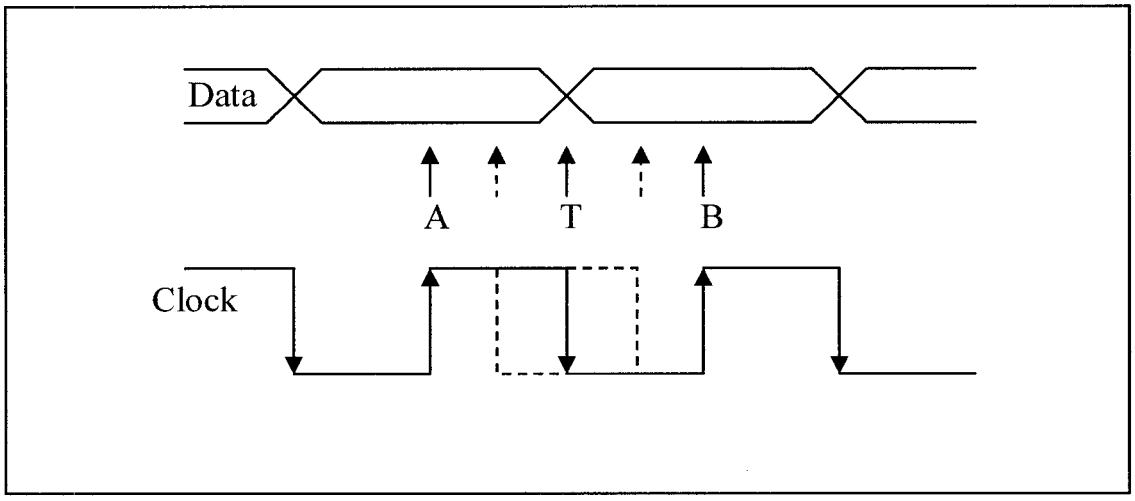


Figure 4.14: Data Sampling Operation

If the same logic level is recorded at both A and B, then a transition has not occurred and no action is taken regardless of the logic level of T. However, if the logic levels of A and B are different, a transition has occurred and the phase detector uses the logic level of T

to determine whether the clock was too early or too late with respect to the data transition. If the logic levels of A and T are the same, but different from B, then the clock was too early and needs to be slowed down. The Alexander phase detector then generates a down flag, which right shifts the register one bit to increase the delay time of the delay line.

On the other hand, if the logic levels of B and T are the same, but different from A, the clock was too late and needs to be speeded up to achieve synchronization. The phase detector generates an up flag which left shift the register one bit to decrease the delay time of the delay line. When sampling the data at point A, the sample is always in the centre of the eye pattern when the data and clock signals are in phase (locked). The values recorded at point A are taken as the retrieved data.

CHAPTER 5
THE IMPLEMENTATION OF ENCRYPTION DES ALGORITHM

5.1 Review

The Data Encryption Standard (DES) block provides data security before transporting payload data within the ATM system. Data transport is streamlined with no impact on the speed of operation of any network transport element. Its implementation is based on the circuit architecture of using a single instantaneous stage, whereby the data is processed through this stage 16 times. The advantages of this proposed design are cost effectiveness and hardware-efficiency, simplicity and fast computation of the DES algorithm. The performance of the encryption block is also very fast coupled with low power consumption when used as a pipelined structure for data encryption.

The advent of Field-Programmable Gate Array (FPGA) technology has created exciting new opportunities for prototyping designs. This chapter describes an implementation of the DES algorithm using FPGA technology. The DES block is used in secure communication networks, where its input data is an MPEG-2 data stream. Output data is an encrypted stream, where the data rate is up to 20 Mbps.

5.2 Architecture of the Coding DES Algorithm

Since the hardware requirements of each stage are identical, previous implementations have used a single instantaneous stage, whereby the data is processed through this stage

16 times [16]. Two authors [30], [35], respectively, described a gate-level design in which the stages were unrolled into a pipeline, and performed by 16 separate instantiations of the hardware, increasing the throughput accordingly.

Our implementation of the DES block uses a single stage. The data passes through this stage 16 times. This technique takes the unrolled structure of the pipeline which would require 16 separate stages and reuse the stage to obtain a rolled structure of pipeline using a single stage. The DES block can continually perform input operations, DES computation and output operations. As a result, the pipeline can simultaneously contain three operations. Thus, there is no impact on the speed of operation of an ATM/SONET transport system. In addition, this DES block is simpler and has a faster data rate than previous designs. Figure 5.1 shows the functional units of the implemented DES algorithm.

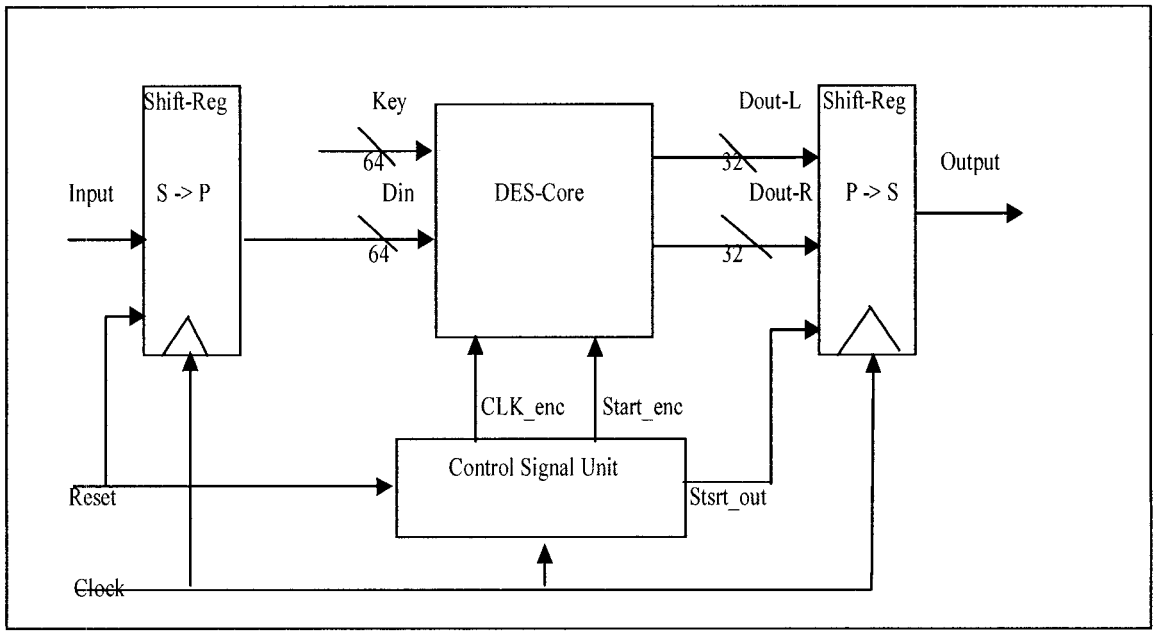


Figure 5.1: Functional Block Diagram of Implemented DES Algorithm

This design consists of four functional blocks:

- Shift register-64 bits (serial-to-parallel)
- Shift register-64 bits (parallel-to-serial)
- DES-Core (computing with DES algorithm)
- Control Signal Unit (generates individual signals for different operations)

Figure 5.2 shows the architecture of one stage of a DES Core. It includes ten blocks as below:

- **FSM** is a loop state signals generator. It creates 16 different state signals to support data calculate repeated16 times.
- **MUX (2->1)** is a two selection one multiplexer. The input data of DES-Core is validity from external (Din) or internal (L-Reg and R-Reg).
- **K-Jumper** function obtains 16 effective keys. It selects 56 bits from 64 bits user input Key, these bits are shifted, and then 48 bits are selected from the 56 bits as DES algorithm.
- **MUX (16->1)** is a sixteen selection one multiplexer. It determinates which 56-bit key is valid in each repeated calculation.
- **R-Jumper** achieves data expansion permutation (EP) processing.
- **XOR-48/32** performances 48/32 bits exclusive-or calculation.
- **S-box** achieves S-Box substitution operation as DES algorithm.
- **L/R-Reg** like a buffer, storage data then send data to the next calculation or output.

The bold lines represent the feedback paths, used 16 times during data processed.

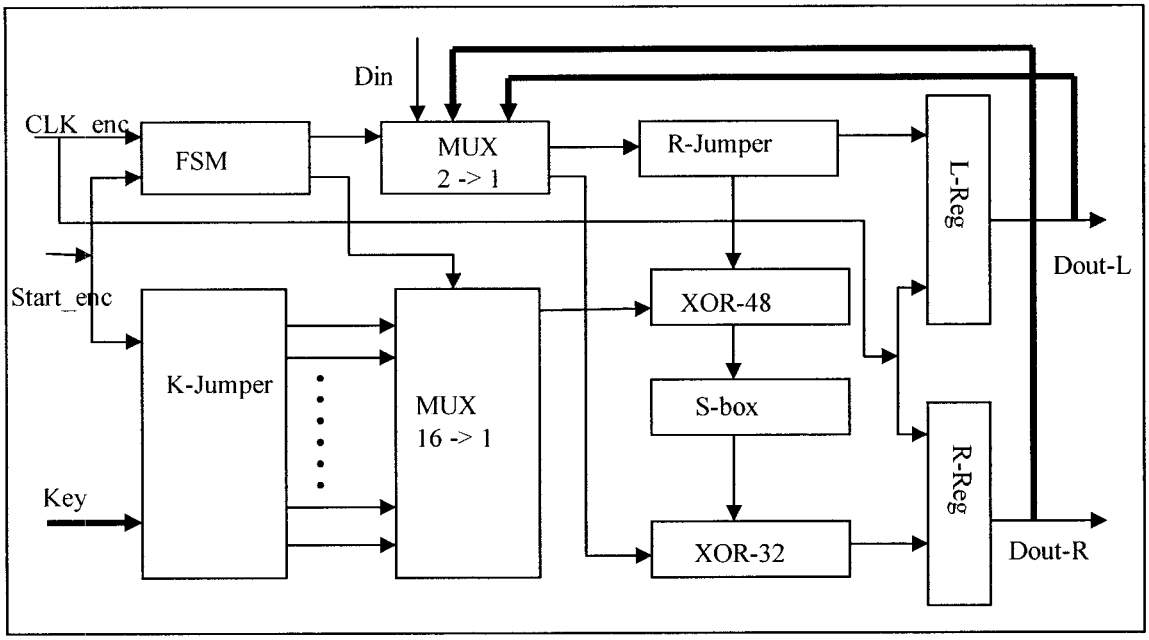


Figure 5.2: Architecture of DES Core

The pipeline techniques are designed to improve system throughput, usually at the instruction level or functional level. The techniques depend on concurrent execution of different functional units of a computation process. For the implementation of the encryption DES algorithm using this technique, the process is divided into three phases:

- Input-data
- DES-computing
- Output-data

The sequence of the three phases is shown at the top of Figure 5.3. Three processes are in the pipeline. In the first time period, input-data processing is performed. In the second time period, DES-computation and the next input-data are both processing data. In the third time period, the output processing, second DES-computation processing, and third input processing are performed simultaneously until the Reset signal is set to "0".

There are two clocks in the pipeline: *CLK_64* signal is used for data input and output, each cycle inputs or outputs a bit data. *CLK_32* signal which is half the frequency of *CLK_64* signal is used for DES computation, which takes 16-cycles. The other 16 cycles of *CLK_32* signal are used to insert 16 wait states. Sample waveforms of the control signals from the control signal unit are shown at the bottom of Figure 5.3.

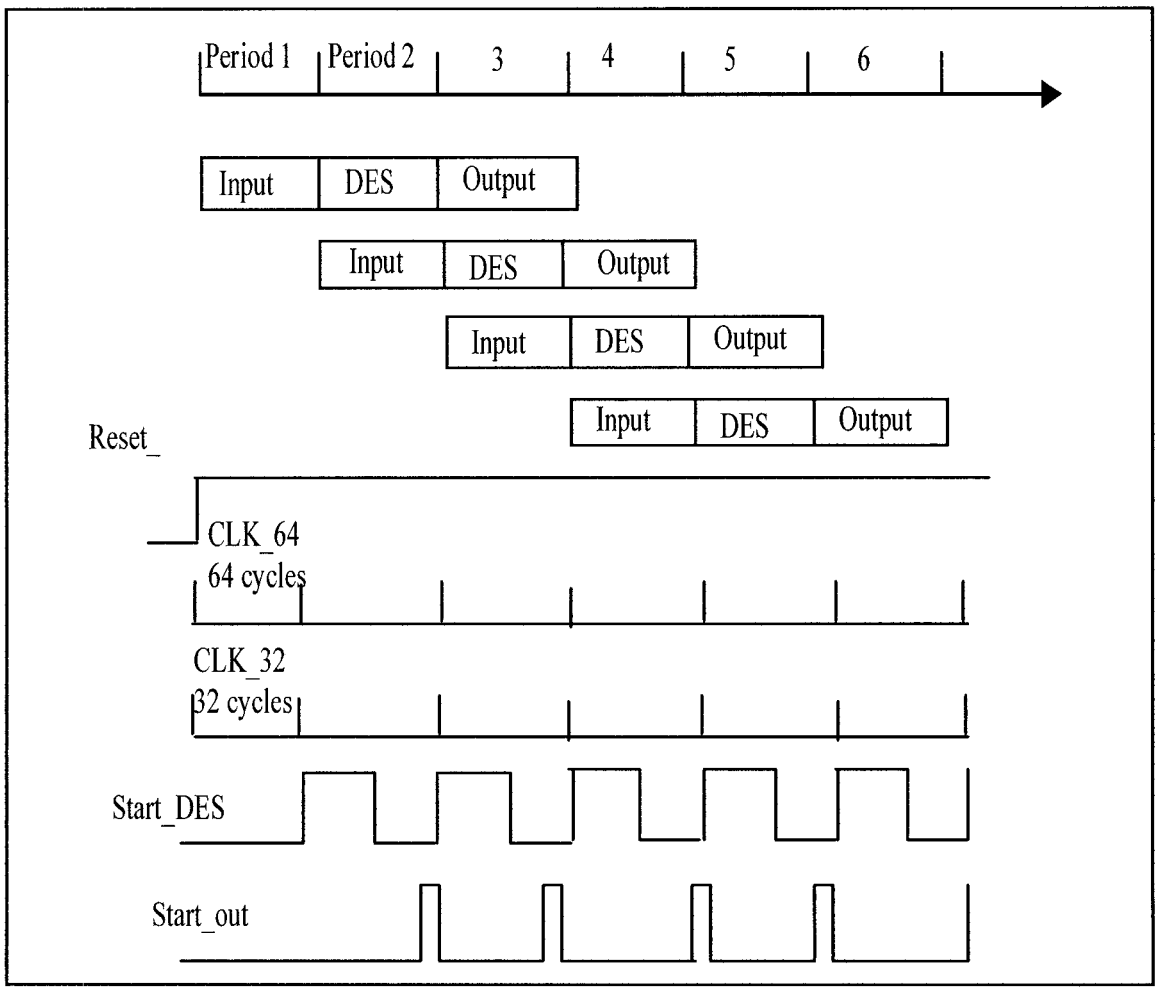


Figure 5.3: Clock and Control Signal for Pipelining Data Processing

A top-down VHDL coding method was used to create the models. The VHDL description of the behavioural design is hierarchical. Figure 5.4 shows the partitioning of the design into VHDL modules.

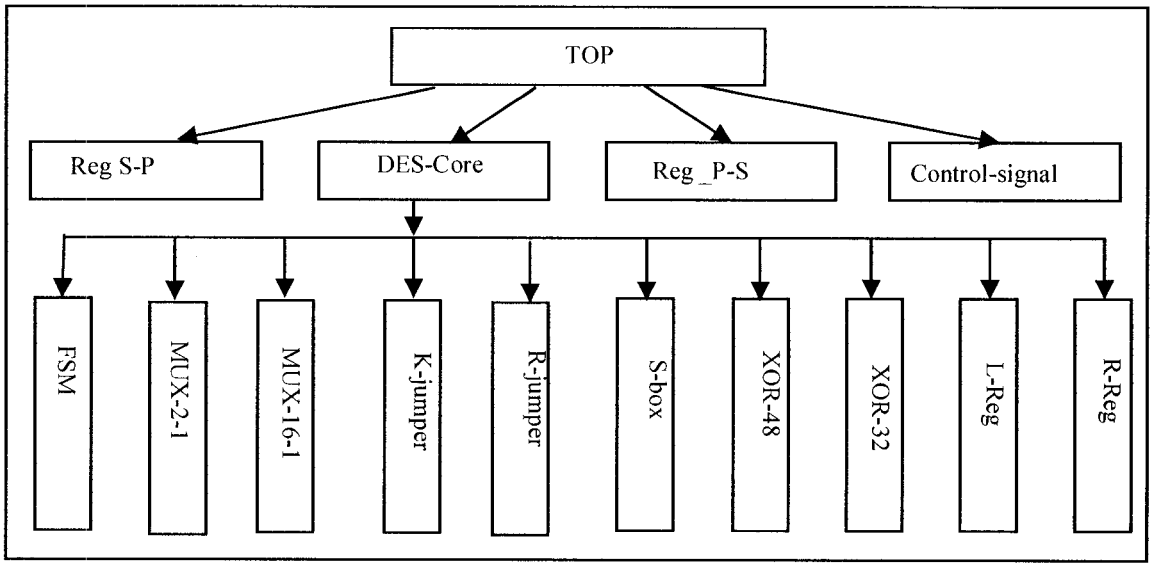


Figure 5.4: Diagram for the Coding of DES Algorithm with VHDL

5.3 Software-Based Logic Verification

The goal of the verification strategy is to efficiently find out any unknown bugs in the design. These bugs may exist anywhere in the process flow. This section focuses on logic verification without time verification.

To precisely verify the logic relationship of the output data from a DES block, a software-based method for automatic verification is used (Figure 5.5). It is based on comparing the result from the software and the hardware. The random key file is a C++ file which generate random 64-bit key by C++ function. The tester program is similar to an Exclusive-OR (XOR) gate. If the input data from the software file and a file generated

by the hardware are the same, then the output of the tester program is 0; otherwise the output is 1. The goal of this testing method is to reduce errors due to the viewing the waveforms by person (Appendix E for further details).

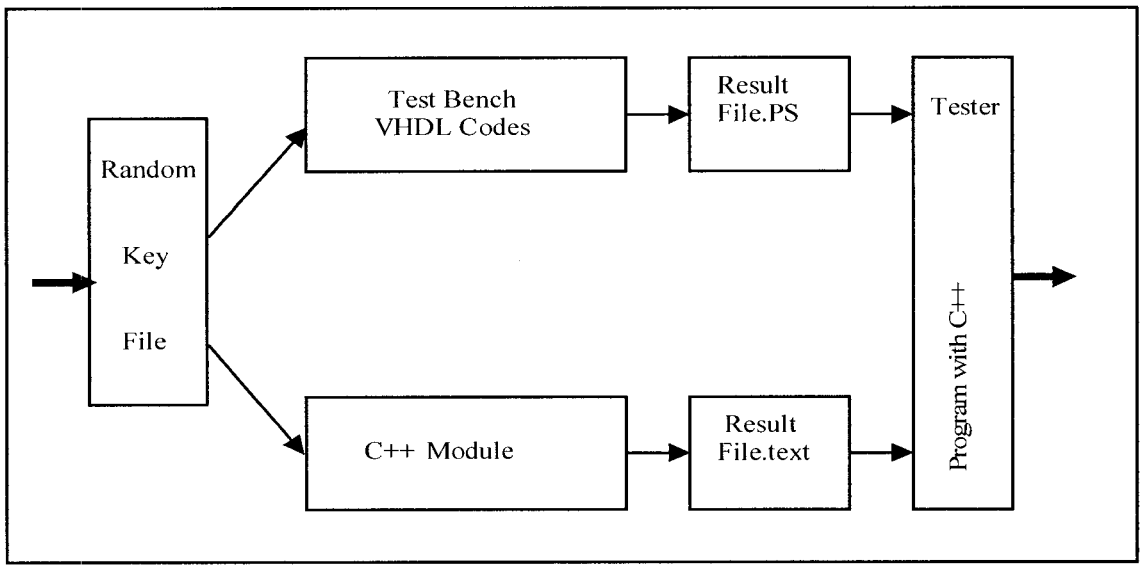


Figure 5.5: Software-Based Method for Automatic Verification

5.4 Prototyping Design for DES Algorithm using an FPGA

The Xilinx XC4000 series devices offer on-chip, edge-triggered, and dual-port RAM clock enables on the I/O flip-flops, and wide-input decoders. These devices provide versatility for many applications, especially those involving RAMs. Design cycles are faster due to a combination of increased routing resources and the use of more sophisticated software tools and techniques.

Xilinx user-programmable gate arrays include two major configurable elements:

- Configurable Logic Blocks (CLBs): These blocks provide the functional elements for constructing the user's logic.

- Input/output Blocks (IOBs): These blocks provide the interface between the package pins and internal signal lines.

Programmable interconnect resources provide routing paths to connect the inputs and outputs of these configurable elements to the appropriate networks. The functionality of each block is customized during configuration by programming internal static memory cell within the block. The values stored in these memory cells determine the logic functions and interconnections implemented in the FPGA.

The XC4000 series is an SRAM-based FPGA that contains logic blocks based on a combination of memory look-up tables and dedicated registers. Considering our design, 192 CLBs registers are required to implement the L-register, the R-register and the serial-to-parallel transfer function. Additional CLBs are needed for the K-jumper, R-jumper, S-box, some counters and the FSM. The XC4028EX provided sufficient resources to meet these requirements.

5.4.1 FPGA Design Flow and Report

Our design intended to be prototyped in Xilinx FPGA is divided into four parts [10, 40]. Part I deals with the VHDL simulation using the Synopsys VHDL System Simulator (VSS). Part II focuses on the logic synthesis using the Synopsys Design Compiler and Design Analyzer tools. Design Compiler shell scripts are used rather than the alternative Design Analyzer Graphical User Interface method of synthesizing a design. This decision is based on the convenience of shell scripts: they may be executed from the command line (negating the need for a graphics workstation terminal) and they may be readily

modified for other designs. Part III concerns the system implementation using the Xilinx Design Manager. In this section, the netlist file obtained as a result of the synthesis step (performed in Part II) is converted into physical hardware. During the last part (Part IV), the netlist file is downloaded into the Xilinx FPGA Demonstration Board. The FPGA Design Flow is depicted in Figure 5.6.

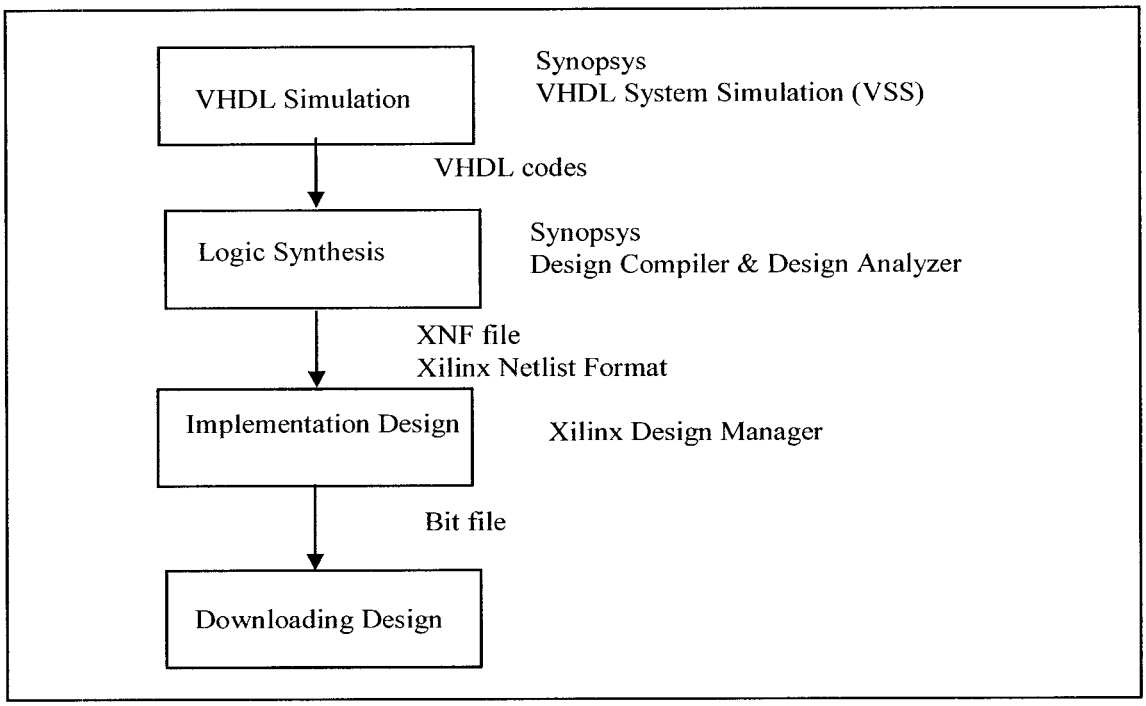


Figure 5.6: The FPGA Design Flow

CHAPTER 6
RESULTS

In this chapter, three sets of test results are described and explained. The first set of tests verifies the operation of the Multiple-High-Frequency SCG using Synopsys and Cadence simulation tools. The second set of tests verifies the ICDPMLIU chip (Multiple-High-Frequency SCG). Also, the simulation results and chip testing results are analysed and compared. Finally, the third set of tests verifies the implementation of the DES encryption algorithm on an FPGA using Mentor Graphics and Xilinx's Design Manager Simulation tools.

6.1 Simulation Result of the Multiple-High-Frequency SCG

The design of the Multiple-High-Frequency SCG is divided into two levels. The top level design includes coding the design in VHDL from the RTL to the gate level and the schematic design at gate level. The second level is the layout design, which includes both auto-layout and custom-layout. In each level, simulation tests are performed. A summary of design and the simulation tools used at each level are described in Table 6.1.

Two different methods were used to perform the simulation. In the first method the load value is set to "1" on Q6 or Q4 which is not centred on Q5 in USR block. During the simulation of the Multiple-High-Frequency SCG, the load value "1" is moved to Q5 step by step. For example, signal Q6 set "1", the other signals Qx(x = 1, 2, 3, 4, 5, 7, 8, 9) set "0" when simulation start, then signal Q6 changes to "0" and signal Q5 changes to"1"

after 50 ns (Figure 6.1). This means that the load value "1" is moving toward Q5, when the *D_CLK* signal is not locked with *S_CLK* signal.

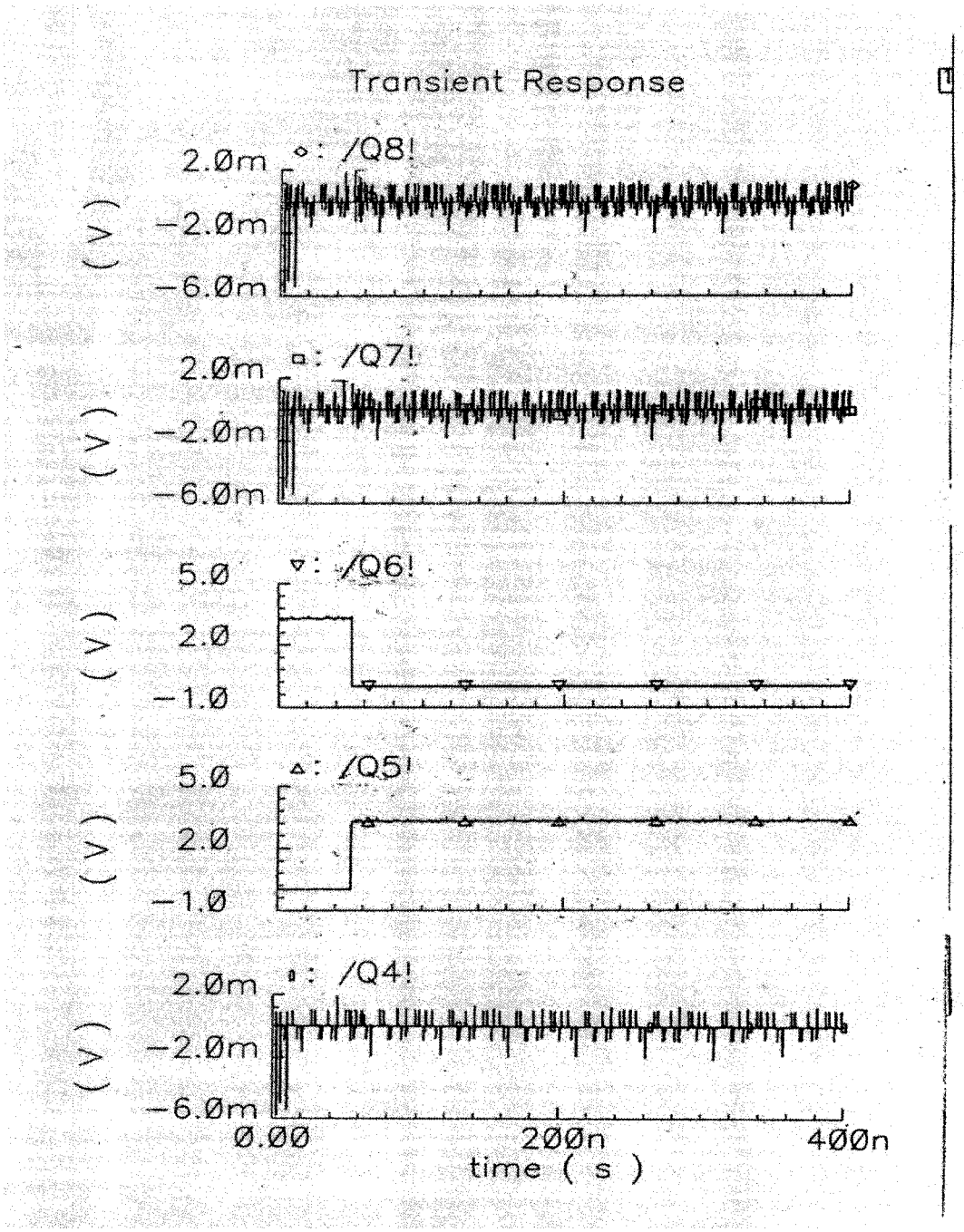


Figure 6.1: Simulation of the Activity of Signals Q4~Q8

Another method is to set the simulation temperature so it is not equal to a normal 27C°. The temperature changes range between -10C°to 50C°. To see the simulation results of the Multiple-High-Frequency SCG, a load value of "1" in USR block is moved to Qx (where x = 1, 2, 3 or 7, 8, 9) step by step.

Table 6.1: Summary of Design Level and Simulation Tools for chip ICDPMLIU

Design Level		Block Simulation	System Simulation	Results in Appendices
Function	RTL to Gate Level (Synopsys)	**USR** Simulation at RTL level (VSS), Synthesis (DC), Simulation at gate level (Verilog-XL)	Cadence (Spectre)	**D-1**: Simulation of SCG with Cadence Spectre
	Schematic Design (Cadence)	**DL, PD** and **FM** simulation (Spectre)		
Layout	Customer Layout (Cadence)	Inverter for Fixed DL (Spectre)	Cadence (Spectre)	**C-1**: Whole chip layout for Multiple-High-Frequency Synchronous Clock Generator (SCG) **C-2**: Layout for SCG core **C-3**: Manual layout for bigger size inverter
	Auto_layout (Cadence)	Whole Chip		

The input frequency of ICDPMLIU chip is 19.44 MHz, the output frequency on stage-1 is 38.88MHz when input signal is made one time multiplication; the output frequency on stage-2 is 77.67 MHz when input signal is made two times multiplication; the output frequency on stage-3 is 155.52 MHz when input signal is made three times multiplication. In short, one cycle is on the input of MF, two cycles are on the output stage-1, four cycles are on the output stage-2, eight cycles are on the output of MF (stage-3). The simulation result by Cadence Spectre is shown in Figure 6.2.

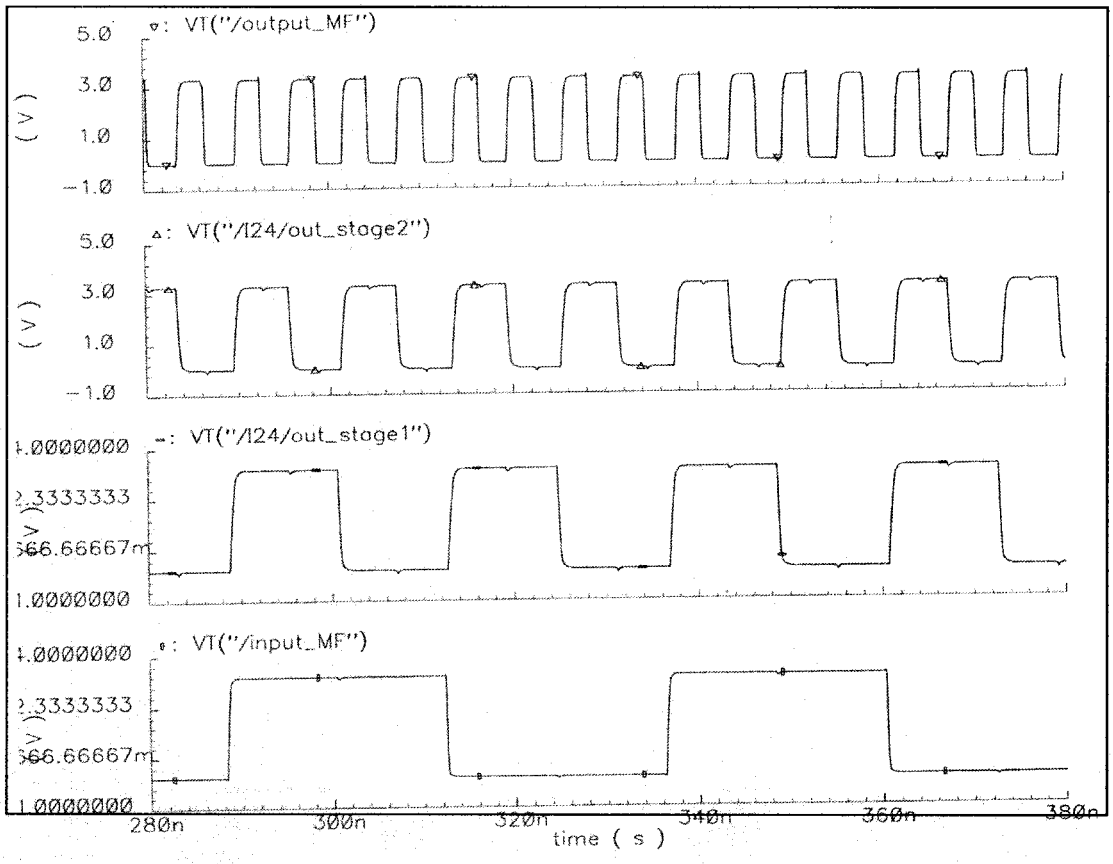


Figure 6.2: The Simulation Results of ICDPMLIU Chip

6.2 Testing of the Fabricated ICDPMLIU Chip

The Multiple-High-Frequency Synchronous Clock Generator (SCG) is fabricated on a chip (ICDPMLIU) using a CMOS 0.35-um process by TSMC (Taiwan Semiconductor Manufacturing Company). As shown in Figure 6.3, the chip testing environment consists of the following equipments:

- DC Power Supply: Model PC-3030, with 5V fixed voltage.
- Signal Generator: 8111 A Pulse / Function Generator, max frequency 20 MHz.
- Oscilloscope: 54616B 2Gsample/s 500 MHz.

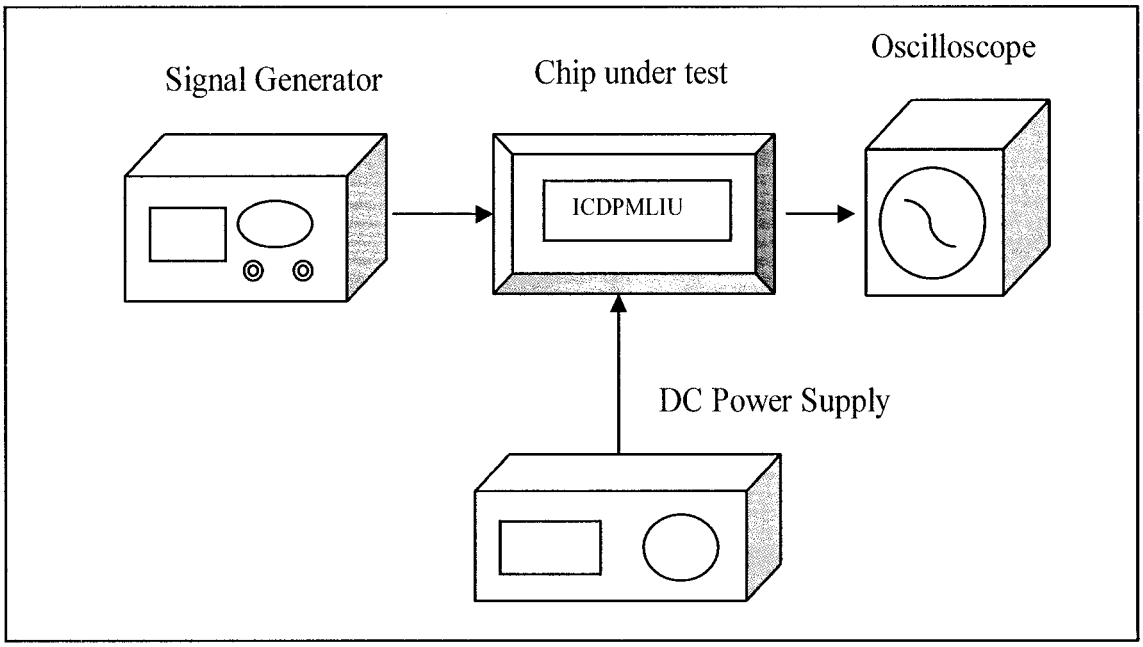


Figure 6.3: The Chip Testing Environment

The chip test results are shown in Figure 6.4 (Input and Output signals on stage-1) and Figure 6.5 (Input and Output signals on stage-3).

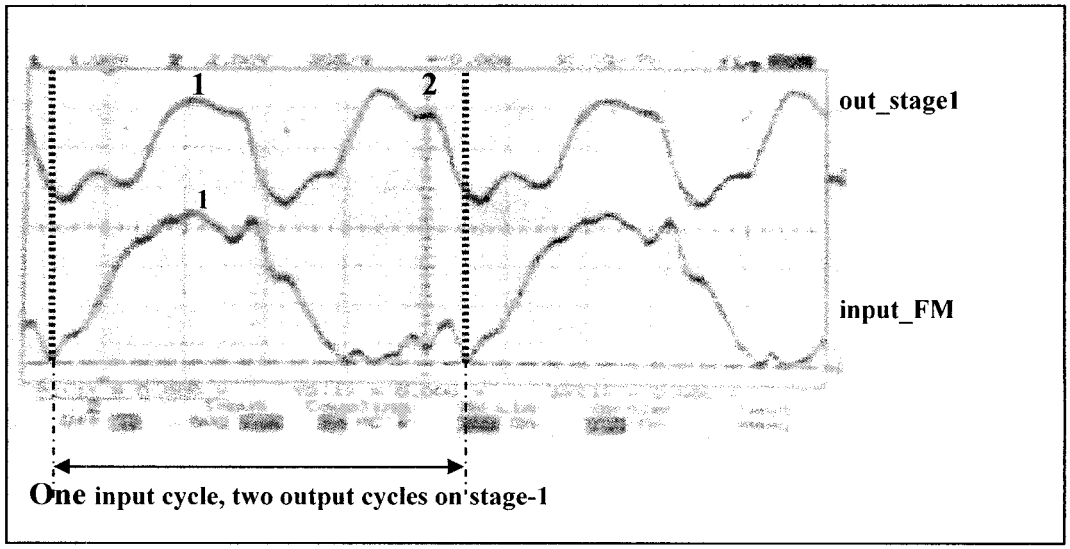


Figure 6.4: Input and Output Signals on Stage-1

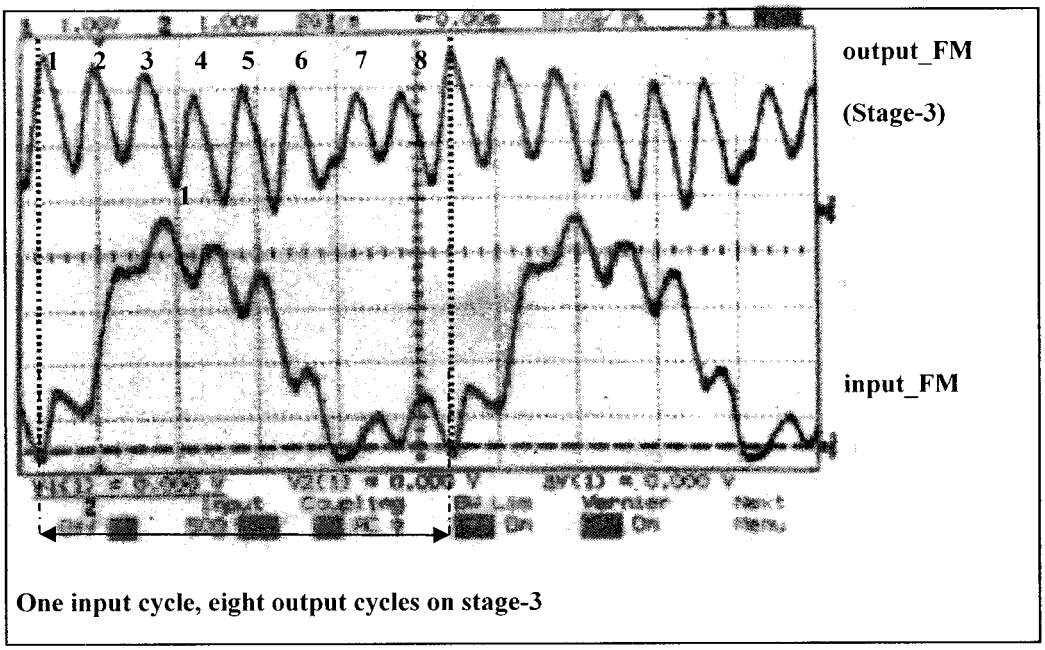


Figure 6.5: Input and Output Signals on Stage-3

A breadboard is used for testing the chip. The waveform pictures of Figure 6.4 and 6.5 show some noise interference in both the input and the output. This noise is probably caused by long connection wires, poor ground connection and low quality of the used power supply. Also a printed circuit board was not available to improve these measurements. However there is no extra pulse at the chip output, there is one cycle in the input of chip and there are eight cycles in the third stage output of chip. This result is correct when considering the limits of testing conditions. Thus, the operation of the ICDPMLIU chip meets our expectation. More accurate results could be obtained if more attention can be paid to the measurement which necessitates larger testing periods.

6.3 Verification of the Implementation of the DES Algorithm

To verify the implementation of the DES algorithm, three aspects of the design are considered.

First, a logic checker module is coded using C++. The function of the logic testing is described in chapter 5 (Software-Based Logic Verification). Two types of data are used: a randomly selected input key, K, of 64 bits and the payload data. These data are passed through the DES algorithm's VHDL module and C++ module individually. The results for each set of output are then compared. For the implementation of the encrypted DES algorithm coded with VHDL created for this thesis, the logic testing produced correct results as explained in Appendix A-4: DES algorithm's VHDL module and Appendix E-1: DES algorithm's C++ module.

Second, a real time simulation at the RTL level using Mentor Graphic simulation tool is performed. This test focuses on the verification of the VHDL module of the DES encryption algorithm. This result confirms the proper operation within the pipeline structure (Figure 6-6: Simulation of DES implementation with Mentor Graphic). The signal "*start_enc_sig*" uses three frames: input data, computation data and output data. The input data stream passes through these three frames of the DES algorithm module. When the first 64-bit data is in the output frame, the second 64-bit data is in the computation frame and the third 64-bit data is in the input frame.

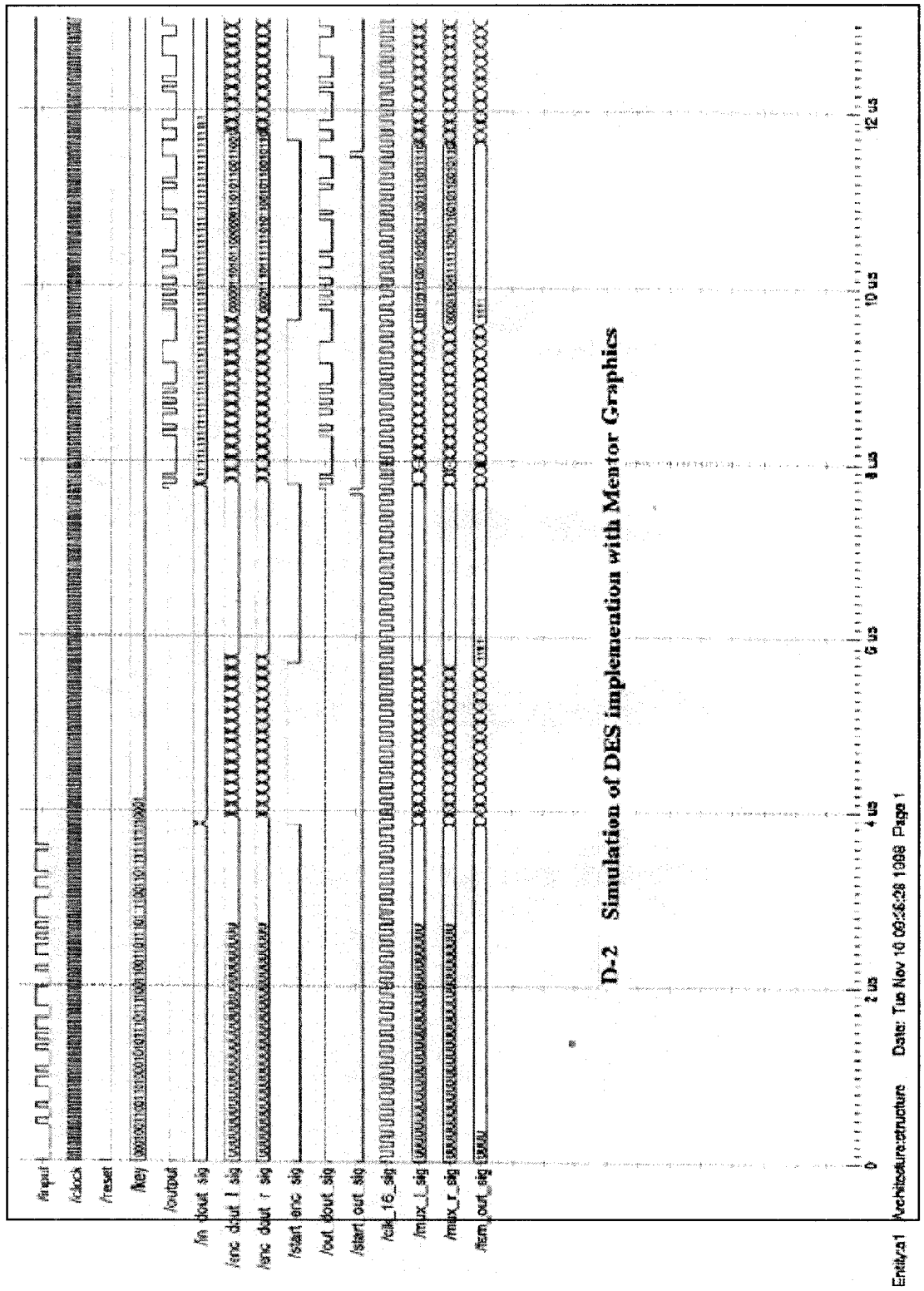


Figure 6.6: Simulation of DES implementation with Mentor Graphic

Finally, a prototype of the DES algorithm module is implemented using Xilinx XC4028-EX device. The script file for the DES algorithm block synthesis design on an XC4028-EX device is in appendix A-6. The timing report and mapping report are in the appendix D-3 and D-4. There is a message report "Slack (MET)" in the timing report in appendix D-3. This positive number indicates that the longest signal path in the multiplier satisfies the clock period constraints. This FPGA mapping uses 707 out of 1024 CLBs which is 69% of the total CLBs and uses 58 out of 256 bonded IOBs which is 23% of the total IOBs. In general, the number of CLBs should not be more than 75% of all bonded IOBs, so the XC4028-EX device is an excellent choice for mapping this design.

Only the work on Part I through Part III is finished in this thesis. Satisfactory results were obtained with the design utilizing, 69% of CLBs and 8% of IOB pads in the XC4028EX FPGA [40]. For more information, see Appendix D-3 and D-4.

CHAPTER 7
CONCLUSION AND FUTURE WORKS

This thesis demonstrates an interface functional block to convert MPEG-2 encrypted data for transport over an ATM/SONET network. The basic ideas, the key technical issues, and future work on this topic are discussed in this chapter.

The design and test reported in this master thesis prove that new circuit technique based on an ADDLL and an MDA to support multiple-stage synchronization on a system are realizable. The sample chip, Multiple-High-Frequency Synchronous Clock Generator, and its whole development process from design, fabrication, and testing, was successful. These results prove that using an ADDLL technique and an MDA to replace the PLL technique for multiple-stage synchronous system on-chip is a promising design option. The benefits include the ability to use purely digital components for locking the transmission clock and the ability to achieve timing recovery in the receiving section. This solution not only overcomes the problem inherent with analog components (passive capacitors and resistors), but also uses a simple process for the design and fabrication of complex synchronous circuits.

Concerning the architecture of the data transport over ATM/SONET, the design of an interface to encrypt MPEG-2 streams to ATM cells was presented in this master thesis. The dual-port RAM-based FIFO is the key part of this interface. The main feature of this FIFO is that its control is simple because the read and write controls are non-correlated. Therefore, read and write data to or from the FIFO can occur simultaneously without introducing wait states. This development makes the FSM's operation in this interface a

practical solution. The implementation of this interface is an important development, only the sample FIFO coding is completed for this thesis work due to time and facility limitations.

On the other hand, the implementation of a DES encryption algorithm on an FPGA is proposed, as well as the functional verification and timing testing during this master thesis research. The data is processed into encrypted data by streamlining the data without impacting the operation or speed of any network transport mechanism. The circuit architecture consists of one instantaneous stage and the data is processed through this state 16 times. The whole process for data input, computing, and data output uses a pipelined technique. The performance of the VHDL model for a DES encryption algorithm is verified using the Xilinx Design Manager Tool (XC4028EX). A Xilinx FPGA Demonstration board was not used due to time limitation. However, this master thesis work proved that this proposed design is cost effective and hardware-efficient.

Finally, our recommendation for future works can be summarized as follow:

In order to apply the ADDLL method for timing recovery, a related diagram on this topic is discussed on chapter 4. However, some detailed structures and functions, such as the control of Delay Chain Oscillator using a Shift Register, the sampling of incoming data when the data patterns change (exchanging "0" and "1"), and comparing two signals in an Alexander Phase Detector and so on, requires more research.

Next, to provide more flexibility for the user, a PCI or USB interface for this system can be added. Some reference source codes for the PCI/USB interface can be found from a

vendor. The key task remaining would be to create the connection between this interface, as described in chapter 2, and the PCI/USB interface.

In addition, fast technological advance in ASIC design allows the implementation of full system-on-chip using purely digital components. A more extensive analysis is required for a real system-level simulation, which would include the integration of the clock generator and the function blocks together.

BIBLIOGRAPHY

1. Alles, A, "ATM Internetworking," ATM Product Line Manager, Cisco Systems, Inc. 1995.
2. American National Standards Institute (ANSI) Inc., "American National Standard - Data Encryption Algorithm (X3.92-1981)," 1981.
3. Applied Micro Circuits Corporation, "The S3019 SONET/SDH/ATM OC-3/12 Transceiver", March, 2000.
4. ATM Forum AF-PHY-0086.000, "Inverse Multiplexing for ATM (IMA) Specification Version 1.0," July, 1997.
5. Atsushi, A, "A 256-mb SDRAM Using a Register-Controlled Digital DLL," IEEE Journal Solid-State Circuits, Volume 32, pp. 1728-1732, November, 1997.
6. Baker, R. J., Harry W. L., and David E. B., "CMOS Circuit Design, Layout and Simulation," IEEE Press Series on Microelectronic Systems, 1997.
7. Beauchamp, K. G. and Poo, G. S "Computer Communications, Third Edition," International Thomson Computer Press, 1995.
8. Brugel, H., and Driessen, P. F., "Variable bandwidth DPLL bit synchronizer with rapid acquisition implemented as a finite state machine", IEEE Trans, Communication, vol. 44, no. 5, pp. 557-561, May 1996.
9. Canadian Microelectronics Corporation, "Instruction on Basic Digital IC Design Flow, from RTL Description to Completed CMOS Design, Using Cadence (97A) and Synopsys", 1998

10. Canadian Microelectronics Corporation, "Digital Logic Synthesis Using Synopsys and Xilinx," 1998.

11. Chiang J. S., and Chen, K. Y., "The Design of an All-Digital Phase-Locked Loop with Small DCO Hardware and Fast Phase Lock", IEEE J. Transactions on Circuits and Systems-II: Analog Digital Signal Processing, vol.46, No. 7, July 1999.

12. Cypress Semiconductor, "The CYS25G0101DX SONET OC-48 Transceiver", 1999.

13. David, H. M., "A Workstation Architecture to Support Multimedia," St. John's College, University of Cambridge, 1993.

14. Dfendovich, A., Afek, Y., Sella, C. and Bbikowsky, Z., "Multi-frequency Zero-jitter Delay-looked Loop," IEEE Journal Solid-State Circuits, Volume 29, pp. 67-70, January, 1994.

15. Dunning, J., Garcia, G., Lundberg, J. and Nuckolls, E.,"An all-digital Phase-locked loop with 50 cycle lock time suitable for high-performance microprocessors," IEEE J. Solid-State Circuit, vol. 30, pp. 412-422, Apr. 1995.

16. Eberle, H., "A High-speed DES Implementation for Network Applications," Technical Report 90, Digital Equipment Corporation Systems Research Centre, September, 1992.

17. Gardener, F. M., "Charge-pump Phase-Look Loops," IEEE Transactions on Communications, COM-28, No. II, PP. 1849-1858, November, 1980.

18. Glykopoulos, G. N., "Design and Implementation of 1.2 Gbit/s ATM Cell Buffer using a Synchronous DRAM Chip," University of Crete, 1998.

19. Goralski, J., "Introduction to ATM Networking," Computing McGraw-Hill, 1995.

20. Gray, P. R. and Meyer, R. G. "Analysis and Design of Analog Integrated Circuits," 3rd edition, John Wiley and Sons, ISBN 0-471-57495-3, 1993.

21. Haykin, S., "An Introduction to Analog and Digital Communications," John Wiley and Sons, ISBN 0_471-85978-8, 1998.

22. Hogge, C. R., "A Self Correcting Clock Recovery Circuit," IEEE Journal Light Wave Technology, Volume LT-3, pp. 1312-1314, December, 1985.

23. Hsu, T. Y., Shieh, B. J., and Lee, C. Y., "An all digital phase-locked loop (ADPLL) based clock recovery circuit," IEEE J. Solid-State Circuit, vol. 34, pp. 1063-1073, Aug.1999.

24. Hsu, T. Y., Wang, C. C., and Lee, C. Y., "Design and Analysis of a Portable High-Speed Clock Generator" IEEE J. Transactions on Circuits and Systems-II: Analog Digital Signal Processing, vol.48, No. 4, Apr. 2001.

25. IBM International Technical Support Organization, "Asynchronous transfer Mode, Technical Overview", October, 1995.

26. Integrated Device Technology, "FIFO Applications GUIDE," Inc., 1999.

27. Lin F., Miller, J. and Baker, R. J.,"A Register-Controlled Symmetrical DLL for Double-Data rate DRAM," IEEE Journal Solid-State Circuits, April, 1999.

28. Mijuskovic, D., Bayer, M., Chomicz, T., Garg, N., James, F., McEntarfer, P., and Porter, J., "Cell-based fully integrated CMOS frequency synthesizer," IEEE J. Solid-State Circuit, vol. 29, pp. 271-279, Mar.1994.

29. Mitchell, L., Pennebaker, B., Fogg, E., and Legall, J. "MPEG Video Compression Standard," International Thompson Publishing, 1997.

30. Michael, W., "Efficient DES Key Search," Technical Report TR-244, School of Computer Science, Carleton University, May, 1994.

31. Monolithic, B. R., "Phase Locked Loop and Clock Recover Circuits," IEEE Press, ISBN 0-7830-1149-3, 1996.

32. Oliver C., "Essentials of ATM Network and Services," Addison Wesley Longman, Inc. 1997.

33. PMC-Sierra, Inc., "ATM Network Interface Card, Technical Overview", 1995.

34. Philips Semiconductors, "Data Sheet OQ2541HP: OQ2541U SDH/SONET Data and Clock Recovery Unit STM1/4/16 OC3/12/48 GE," Product Specification, 1999.

35. Schaffer, T., Alan G., Srisai R. and Paul F., "A Flip-Chip Implementation of the Data Encryption Standard (DES)," IEEE Multi-Chip Module Conference, 1997.

36. Sterbentz, P. G., "ATM/B-IDSN Tutorial Notes," GTE Telecommunications Research Laboratory, 1994.

37. Taiwan Semiconductor Manufacturing Company Ltd., "TSMC 0.35 μm Mixed-Mode Polycide 3.3/5V Design Rule," 1999.

38. Wolaver, D. H "Phase-Locked Loop Circuit Design", Englewood Cliffs, NJ: Prentice-Hall, 1991.

39. Xanthaki, Z., "A Memory Controller Access Interleaving over a Single Rambus," University of Crete, 1994.

40. XILINX, Inc., "The Programmable Logic Data Book," 1998.

41. Yang, H. C., Lee, L. K., and Co, R, S., "A low jitter 0.3-165 Mhz CMOS PLL frequency synthesizer for 3 V/5 V operation," IEEE J. Solid-State Circuit, vol. 32, pp. 582-586, Apr. 1997.

42. Young, L. A., Greason, J. K., and Wong, K. L., "A PLL Clock Generator with 5 to 100 MHz of Lock Rang for Microprocessor," IEEE Journal of Solid-State Circuits, Volume SC-27, pp. 1599-1607, November, 1992.

APPENDICES

```
// File Name : FIFO.v
//
// Input ports:
//
// All ports with a suffix "_" are low asserted.
// clk_in          Input clock signal
// clk_out         Output clock signal
// reset_          Reset signal
// data_in         8_bits data into FIFO
// fifo_wd_en      Enable write data into FIFO
// fifo_rd_en      Enable read data from FIFO

// Output ports:

// data_out       8_bits data output from FIFO
// full_flag      Signal indicating when FIFO is full
// empty_flag     Signal indicating when FIFO is empty

`define WIDTH   1       // width of the FIFO
`define DEPTH   376     // depth of the FIFO
`define CWIDTH  9       // counter width is 2 to power CWIDTH > DEPTH

module FIFO (  clk_in,
               clk_out,
               reset_,
               data_in,
               fifo_wd_en,
               fifo_rd_en,


               data_out,
               full_flag,
               empty_flag,
               );

  input                     clk_in;      // input clock signal
  input                     clk_out;     // output clock signal
  input                     reset_;      // low asserted reset signal
  input [(`WIDTH -1):0]     data_in;     // data into FIFO
  input                     fifo_wd_en;  // enable write data into FIFO
  input                     fifo_rd_en;  // enable read data from FIFO


  output [(`WIDTH -1):0]    data_out;    // data output FIFO
  output                    full_flag;   // flag of FIFO full
  output                    empty_flag;  // flag of FIFO empty


  reg                       full_flag;
  reg                       empty_flag;
  reg [(`WIDTH -1):0]       data_out;
  reg [C`CWIDTH-1) :0 ]     counter;  // counter indicates num of data in FIFO
  reg [(`DEPTH -1):0]       rd_ptr;    // current read pointer
  reg [(`DEPTH -1):0]       wr_ptr;    // current write pointer
  wire [(`WIDTH -1):0]      fifo_data_out; // data out from FIFO MemBlk
  wire [(`WIDTH -1):0]      fifo_data_in;  // data into FIFO MemBlk

  wire                      read = fifo_rd_en;
  wire                      write = fifo_wd_en;

  assign data_out = fifo_data_out;
  assign fifo_data_in =  data_in;
```



```
  FIFO_MEM_BLK FSM_FIFO (.mem_clk_in (clk_in),
                         .mem_clk_out (clk_out),
                         .mem_write_ (write),
                         .mem_read_ (read),
                         .mem_wr_addr (wr_ptr),
                         .mem_rd_addr (rd_ptr),
                         .mem_data_out (fifo_data_out),
                         .mem_data_in (fifo_data_in)
                         );

// Initialize and W/R point reset

always @(posedge clk_in)
    begin
       if (reset_ = 0) // reset
         begin
            rd_ptr[(`DEPTH -1):0]  = 0;
            wr_ptr[(`DEPTH -1):0]  = 0;
            counter[(`CWIDTH - 1):0] = 0;
            full_flag = 1'b0;
            empty_flag = 1'b1;
            reset_ = 1'b1;
         end
     else
        begin
          if((wr_ptr = `DEPTH-1) & (rd_ptr != 0))
            wr_ptr[(`DEPTH -1):0] = 0;
          if((rd_ptr = `DEPTH-1) & (wr_ptr != 0))
            rd_ptr[(`DEPTH -1):0]  = 0;
        end

    end // always @ (posedge clk_in )

// FIFO is empty

 always @(posedge clk_in)
    begin
      if(reset_ = 1)
        begin
         if((empty_flag = 1) & (full_flag = 0) & fifo_wr_en = 1));
            begin
              wr_ptr = wr_ptr + 1;
              counter = counter +1;
              empty_flag = 1'b0;
            end
         if((fifo_wr_en = 0) & (fifo_rd_en = 0))
            reset_ = 1'b0;
        end
   end // always @ (posedge clk_in)

// FIFO is full

 always @(posedge clk_in)
    begin
      if(reset_ = 1)
        begin
         if((empty_flag = 0) & (full_flag = 0) & fifo_wr_en = 1));
            if( counter = `DEPTH - 1)
                 full_flag = 1'b1;
```

```
    end // always @ (posedge clk_in)


 // FIFO is reading data

  always @(posedge clk_out)
    begin
       if((fifo_rd_en = 1) & (empty_flag = 0))
         begin
            if(counter < FDEPTH - 1 )
              begin
                rd_ptr <= rd_ptr + 1;
                counter = counter - 1;
              end
            if(counter = 0)
              begin
                 empty_flag = 1'b1;
              end
            if((fifo_wr_en = 0) & (fifo_rd_en = 0))
            reset_ = 1'b0;
         end
    end // always @ (posedge clk_out)


   // FIFO is writing data
   always @(posedge clk_in)
      begin
        if((fifo_wr_en = 1) & (full_flag  = 0))
          begin
             if(counter < DEPTH - 1)
               begin
                 wr_ptr <= wr_ptr + 1;
                 counter = counter + 1;
               end
             if (counter = DEPTH - 1)
                full_flag = 1'b1;
             if((fifo_wr_en = 0) & (fifo_rd_en = 0))
               reset_ = 1'b0;
          end
    end // always @ (posedge clk_in)

endmodule // FIFO



module FIFO_MEM_BLK ( mem_clk,
                      mem_write_,
                      mem_read_,
                      mem_wr_addr,
                      mem_rd_addr,
                      mem_data_in,
                      mem_data_out
                      );

  input mem_clk;       // input clk
  input mem_write_;    // write signal to put data into FIFO
  input mem_read_;     // read signal to output data from FIFO
  input [('CWIDTH -1):0] mem_wr_addr;  // write address
  input [('CWIDTH -1):0] mem_rd_addr;  // read address
  input [('WIDTH -1):0]  mem_data_in;  // data into memory block

  output [('WIDTH -1):0] mem_data_out; // data output from memory block
```



```
  wire [('WIDTH -1):0] mem_data_out;

  reg [('WIDTH -1):0]   FIFO [0:('DEPTH-1)];

  assign mem_data_out = FIFO [mem_rd_addr];

  always @(posedge mem_clk)
    begin
      if(mem_write_ == 1'b0)
        FIFO[mem_wr_addr] <= mem_data_in;
    end // always @ (posedge mem_clk)
endmodule
```

A-2 VHDL codes for universal shirt register (USR)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Good_USR IS
              PORT( CLK          : in  std_logic;
                    Reset        : in  std_logic;
                    S0           : in  std_logic;
                    S1           : in  std_logic;
                    QA           : buffer std_logic;
                    QB           : buffer std_logic;
                    QC           : buffer std_logic;
                    QD           : buffer std_logic;
                    QE           : buffer std_logic;
                    QF           : buffer std_logic;
                    QG           : buffer std_logic;
                    QH           : buffer std_logic;
                    QI           : buffer std_logic;
                    NQA          : buffer std_logic;
                    NQB          : buffer std_logic;
                    NQC          : buffer std_logic;
                    NQD          : buffer std_logic;
                    NQE          : buffer std_logic;
                    NQF          : buffer std_logic;
                    NQG          : buffer std_logic;
                    NQH          : buffer std_logic;
                    NQI          : buffer std_logic);
END Good_USR;

architecture behavior of Good_USR is
begin
   process ( CLK, Reset, S0, S1)
                subtype TWO_BIT is std_logic_vector ( 0 to 1 );
             begin
             if Reset = '1' then
                QA <= '0';
                QB <= '0';
                QC <= '0';
                QD <= '0';
                QE <= '0';
                QF <= '0';
                QG <= '1';
                QH <= '0';
                QI <= '0';

                NQA <= '1';
                NQB <= '1';
                NQC <= '1';
                NQD <= '1';
                NQE <= '1';
                NQF <= '1';
                NQG <= '0';
                NQH <= '1';
                NQI <= '1';
```

```
        elsif (CLK' event and CLK = '1') then
          case TWO_BIT'(S0&S1) is
             when "00" =>
                null;
             when "01" =>
                QA <= QB;
                QB <= QC;
                QC <= QD;
                QD <= QE;
                QE <= QF;
                QF <= QG;
                QG <= QH;
                QH <= QI;
                QI <= QA;

                NQA <= not QB;
                NQB <= not QC;
                NQC <= not QD;
                NQD <= not QE;
                NQE <= not QF;
                NQF <= not QG;
                NQG <= not QH;
                NQH <= not QI;
                NQI <= not QA;

             when "10" =>
                QA <= QI;
                QB <= QA;
                QC <= QB;
                QD <= QC;
                QE <= QD;
                QF <= QE;
                QG <= QF;
                QH <= QG;
                QI <= QH;

                NQA <= not QI;
                NQB <= not QA;
                NQC <= not QB;
                NQD <= not QC;
                NQE <= not QD;
                NQF <= not QE;
                NQG <= not QF;
                NQH <= not QG;
                NQI <= not QH;

             when others =>
                  null;
            end case;
          end if;
    end process;
end behavior;
```

A-3 VHDL codes for universal shirt register (USR)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Good_USR IS
              PORT( CLK          : in  std_logic;
                    Reset        : in  std_logic;
                    S0           : in  std_logic;
                    S1           : in  std_logic;
                    QA           : buffer std_logic;
                    QB           : buffer std_logic;
                    QC           : buffer std_logic;
                    QD           : buffer std_logic;
                    QE           : buffer std_logic;
                    QF           : buffer std_logic;
                    QG           : buffer std_logic;
                    QH           : buffer std_logic;
                    QI           : buffer std_logic;
                    NQA          : buffer std_logic;
                    NQB          : buffer std_logic;
                    NQC          : buffer std_logic;
                    NQD          : buffer std_logic;
                    NQE          : buffer std_logic;
                    NQF          : buffer std_logic;
                    NQG          : buffer std_logic;
                    NQH          : buffer std_logic;
                    NQI          : buffer std_logic);
END Good_USR;

architecture behavior of Good_USR is
begin
   process ( CLK, Reset, S0, S1)
                subtype TWO_BIT is std_logic_vector ( 0 to 1 );
             begin
             if Reset = '1' then
                QA <= '0';
                QB <= '0';
                QC <= '0';
                QD <= '0';
                QE <= '0';
                QF <= '0';
                QG <= '1';
                QH <= '0';
                QI <= '0';

                NQA <= '1';
                NQB <= '1';
                NQC <= '1';
                NQD <= '1';
                NQE <= '1';
                NQF <= '1';
                NQG <= '0';
                NQH <= '1';
                NQI <= '1';
```

```
                    NQH             : buffer std_logic;
                    NQI             : buffer std_logic);
END component;

begin
              U1: Good_USR PORT map (
                    CLK          => CLK,
                    Reset        => Reset,
                    S0           => S0,
                    S1           => S1,
                    QA           => QA,
                    QB           => QB,
                    QC           => QC,
                    QD           => QD,
                    QE          => QE,
                    QF           => QF,
                    QG           => QG,
                    QH           => QH,
                    QI           => QI,
                    NQA          => NQA,
                    NQB          => NQB,
                    NQC          => NQC,
                    NQD          => NQD,
                    NQE          => NQE,
                    NQF          => NQF,
                    NQG          => NQG,
                    NQH          => NQH,
                    NQI          => NQI);

               U2: process
                  begin
                   CLK <= '1','0' after 24 ns;
                              wait for 48 ns;
          end process;

        U3:process
          begin
            Reset <= '0', '1' after 5 ns, '0' after 15 ns;
            S0    <= '1', '0' after 15 ns, '1' after 300 ns;
            S1    <= '1', '0' after 15 ns, '1' after 60 ns, '0' after 200 ns;
            wait for 2000 ns;
          end process;
end;

configuration cfg_good_USR of test_USR is
              for  A1
        end for;
end cfg_good_USR;
```

```
------------------------------------------------------------
/
/ A-5   VHDL codes for encryption with DES algorithm
/
------------------------------------------------------------

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity TOP is
        port (input       : in  std_logic;
              Clock       : in  std_logic;
              reset       : in  std_logic;
              key         : in  std_logic_vector ( 63 downto 0 );
              output      : out std_logic);
end TOP;

architecture structure of TOP is

component IN_D

        PORT( in_din    : IN std_logic;
              CLK       : IN std_logic;
              reset     : IN std_logic;
              flag_dout : in std_logic;
              in_dout   : OUT std_logic_vector (63 downto 0) );
end component;

component MUX_16_1
        port (  MUX_1   : in std_logic_vector ( 47 downto 0 );
                MUX_2   : in std_logic_vector ( 47 downto 0 );
                MUX_3   : in std_logic_vector ( 47 downto 0 );
                MUX_4   : in std_logic_vector ( 47 downto 0 );
                MUX_5   : in std_logic_vector ( 47 downto 0 );
                MUX_6   : in std_logic_vector ( 47 downto 0 );
                MUX_7   : in std_logic_vector ( 47 downto 0 );
                MUX_8   : in std_logic_vector ( 47 downto 0 );
                MUX_9   : in std_logic_vector ( 47 downto 0 );
                MUX_10  : in std_logic_vector ( 47 downto 0 );
                MUX_11  : in std_logic_vector ( 47 downto 0 );
                MUX_12  : in std_logic_vector ( 47 downto 0 );
                MUX_13  : in std_logic_vector ( 47 downto 0 );
                MUX_14  : in std_logic_vector ( 47 downto 0 );
                MUX_15  : in std_logic_vector ( 47 downto 0 );
                MUX_16  : in std_logic_vector ( 47 downto 0 );
                MUX_sel : in std_logic_vector     downto 0 );
                MUX_out : out std_logic_vecto     downto 0 ));
end component;

component KKK
        port ( K_data    : in  std_logic_vector ( 63 downto 0);
               K1        : out std_logic_vector ( 47 downto 0 );
               K2        : out std_logic_vector ( 47 downto 0 );
               K3        : out std_logic_vector ( 47 downto 0 );
               K4        : out std_logic_vector ( 47 downto 0 );
               K5        : out std_logic_vector ( 47 downto 0 );
               K6        : out std_logic_vector ( 47 downto 0 );
               K7        : out std_logic_vector ( 47 downto 0 );
               K8        : out std_logic_vector ( 47 downto 0 );
               K9        : out std_logic_vector ( 47 downto 0 );
               K10       : out std_logic_vector ( 47 downto 0 );
               K11       : out std_logic_vector ( 47 downto 0 );
               K12       : out std_logic_vector ( 47 downto 0 );
               K13       : out std_logic_vector ( 47 downto 0 );
               K14       : out std_logic_vector ( 47 downto 0 );
               K15       : out std_logic_vector ( 47 downto 0 );
               K16       : out std_logic_vector ( 47 downto 0 ));
end component;

component MUX_2_1

        port (  start_M      : in std_logic;
                MUX_Reg_64   : in std_logic_vector ( 63 downto 0 );
                MUX_Reg_L    : in std_logic_vector ( 31 downto 0 );
                MUX_Reg_R    : in std_logic_vector ( 31 downto 0 );
                MUX_sel      : in std_logic_vector ( 3  downto 0 );
                MUX_out_L    : out std_logic_vector ( 31 downto 0 );
                MUX_out_R    : out std_logic_vector ( 31 downto 0 ));

end component;

component FSM
        port (CLK_cycle   : in  std_logic;
              start_FSM   : in  std_logic;
              FSM_out     : out std_logic_vector ( 3 downto 0 ));
end component;

component Reg_32

        port( CLK_Reg     : in  std_logic;
              Reg_in      : in  std_logic_vector ( 31 downto 0 );
              Reg_out     : out std_logic_vector ( 31 downto 0 ));
end component;


component R_Reg

        port( R_in      : in  std_logic_vector ( 31 downto 0 );
              R_out     : out std_logic_vector ( 47 downto 0 );
              R_out_32  : out std_logic_vector ( 31 downto 0 ));
end component;

component XOR_32

        PORT ( A : in  std_logic_vector ( 31 downto 0 );
               B : in  std_logic_vector ( 31 downto 0 );
               C : out std_logic_vector ( 31 downto 0 ));

END component;

component XOR_48

        PORT ( A : in  std_logic_vector ( 47 downto 0 );
               B : in  std_logic_vector ( 47 downto 0 );
               C : out std_logic_vector ( 47 downto 0 ));

END component;

component S_box
```

```
        PORT ( DIN  : in  std_logic_vector ( 47 downto 0 );
               DOUT : out std_logic_vector ( 31 downto 0 ));
END component;

component S_Generater
        port( CLK       : in   std_logic;
              CLK_16    : out  std_logic;
              start_enc : out  std_logic;
              start_out : out  std_logic);
end component;

component OUT_D
        port( out_din       : in std_logic_vector ( 63 downto 0 );
              CLK           : in std_logic;
              start_out     : in std_logic;
              out_dout      : out std_logic);
end component;

signal  in_dout_sig      :  std_logic_vector ( 63 downto 0);
signal  enc_dout_L_sig:  std_logic_vector ( 31 downto 0);
signal  enc_dout_R_sig:  std_logic_vector ( 31 downto 0);
signal  start_enc_sig     :  std_logic;
signal  out_dout_sig  :  std_logic;
signal  start_out_sig     :  std_logic;
signal  CLK_16_sig        :  std_logic;


signal MUX_L_sig         : std_logic_vector ( 31 downto 0 );
signal MUX_R_sig         : std_logic_vector ( 31 downto 0 );
signal FSM_out_sig       : std_logic_vector ( 3  downto 0 );
--signal R_out_sig        : std_logic_vector ( 31 downto 0 );
--signal L_out_sig        : std_logic_vector ( 31 downto 0 );
signal MUX_out_sig       : std_logic_vector ( 47 downto 0 );
signal K1_sig            : std_logic_vector ( 47 downto 0 );
signal K2_sig            : std_logic_vector ( 47 downto 0 );
signal K3_sig            : std_logic_vector ( 47 downto 0 );
signal K4_sig            : std_logic_vector ( 47 downto 0 );
signal K5_sig            : std_logic_vector ( 47 downto 0 );
signal K6_sig            : std_logic_vector ( 47 downto 0 );
signal K7_sig            : std_logic_vector ( 47 downto 0 );
signal K8_sig            : std_logic_vector ( 47 downto 0 );
signal K9_sig            : std_logic_vector ( 47 downto 0 );
signal K10_sig           : std_logic_vector ( 47 downto 0 );
signal K11_sig           : std_logic_vector ( 47 downto 0 );
signal K12_sig           : std_logic_vector ( 47 downto 0 );
signal K13_sig           : std_logic_vector ( 47 downto 0 );
signal K14_sig           : std_logic_vector ( 47 downto 0 );
signal K15_sig           : std_logic_vector ( 47 downto 0 );
signal K16_sig           : std_logic_vector ( 47 downto 0 );

signal   B1_32_sig       : std_logic_vector ( 31 downto 0 );
signal   A1_48_sig       : std_logic_vector ( 47 downto 0 );
signal   B1_48_sig       : std_logic_vector ( 47 downto 0 );
signal   C1_48_sig       : std_logic_vector ( 47 downto 0 );
signal   L_next_sig      : std_logic_vector ( 31 downto 0 );
signal   R_next_sig      : std_logic_vector ( 31 downto 0 );

begin
```



```
U_FSM: FSM port map (
        CLK_cycle   => Clock,
        start_FSM   => start_enc_sig,
        FSM_out     => FSM_out_sig);


U_MUX2: MUX_2_1 port map (
        start_M     => start_enc_sig,
        MUX_Reg_64  => in_dout_sig,
        MUX_Reg_L   => enc_dout_L_sig,
        MUX_Reg_R   => enc_dout_R_sig,
        MUX_sel     => FSM_out_sig,
        MUX_out_L   => MUX_R_sig,
        MUX_out_R   => MUX_L_sig);

U_MUX: MUX_16_1 port map (
        MUX_1   => K1_sig,
        MUX_2   => K2_sig,
        MUX_3   => K3_sig,
        MUX_4   => K4_sig,
        MUX_5   => K5_sig,
        MUX_6   => K6_sig,
        MUX_7   => K7_sig,
        MUX_8   => K8_sig,
        MUX_9   => K9_sig,
        MUX_10  => K10_sig,
        MUX_11  => K11_sig,
        MUX_12  => K12_sig,
        MUX_13  => K13_sig,
        MUX_14   => K14_sig,
        MUX_15   => K15_sig,
        MUX_16   => K16_sig,
        MUX_sel  => FSM_out_sig,
        MUX_out  => MUX_out_sig);

U_KKK: KKK port map (
        K_data   => key,
        K1       => K1_sig,
        K2       => K2_sig,
        K3       => K3_sig,
        K4       => K4_sig,
        K5       => K5_sig,
        K6       => K6_sig,
        K7       => K7_sig,
        K8       => K8_sig,
        K9       => K9_sig,
        K10      => K10_sig,
        K11      => K11_sig,
        K12      => K12_sig,
        K13      => K13_sig,
        K14      => K14_sig,
        K15      => K15_sig,
        K16      => K16_sig);

U_R:     Reg_32 port map (
        CLK_Reg     => CLK_16_sig,
        Reg_in      => L_next_sig,
        Reg_out     => enc_dout_R_sig);

U_L:     Reg_32 port map (
```

```
            CLK_Reg     => CLK_16_sig,
            Reg_in      => R_next_sig,
            Reg_out     => enc_dout_L_sig);

      U_R1 : R_Reg port map (
            R_in        => MUX_R_sig,
            R_out_32    => L_next_sig,
            R_out       => A1_48_sig);

      U_XOR1_32 : XOR_32 port map (
            A    => MUX_L_sig,
            B    => B1_32_sig,
            C    => R_next_sig );

      U_XOR1_48 : XOR_48 port map (
            A    => A1_48_sig,
            B    => MUX_out_sig,
            C    => C1_48_sig);

      U_S1_box : S_box port map (
             DIN    => C1_48_sig,
             DOUT   => B1_32_sig );


      U_IN_D : IN_D port map(

            in_din      => input,
            CLK         => Clock,
            reset       => reset,
            flag_dout=> start_enc_sig,
            in_dout     => in_dout_sig);

      U_S : S_Generater port map (
            CLK .         => Clock,
            CLK_16        => CLK_16_sig,
            start_out     => start_out_sig,
            start_enc     => start_enc_sig );

      U_OUT_D : OUT_D port map (
            out_din ( 31 downto 0 ) => enc_dout_R_sig,
            out_din ( 63 downto 32) => enc_dout_L_sig,
            CLK             => Clock,
            start_out       => start_out_sig,
            out_dout        => out_dout_sig );

            output   <= out_dout_sig;


end structure;

-------------------------------------------------------------------

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY IN_D IS
      PORT( in_din      : IN std_logic;
            CLK         : IN std_logic;
```



```
            reset       : IN std_logic;
            flag_dout: in  std_logic;
            in_dout     : OUT std_logic_vector (63 downto 0) );
END IN_D;

ARCHITECTURE comport OF IN_D IS
   TYPE memor is array (63 downto 0) OF std_logic;
   SIGNAL data_inter : memor;
BEGIN
      dec_process : PROCESS
         VARIABLE data1 : memor;
      BEGIN
            wait until( CLK' EVENT AND CLK = '1');
               IF ( reset = '0' ) THEN
                  FOR I IN 62 downto 0 LOOP
                     data1(I+1) := data_inter(I);
                  END LOOP;
                  data1(0) := in_din;
                  data_inter <= data1;
               END IF;
      END PROCESS dec_process;

      out_process: process(flag_dout)
      begin
      if (flag_dout = '1') then
      in_dout(63) <= data_inter(6);
      in_dout(62) <= data_inter(14);
      in_dout(61) <= data_inter(22);
      in_dout(60) <= data_inter(30);
      in_dout(59) <= data_inter(38);
      in_dout(58) <= data_inter(46);
      in_dout(57) <= data_inter(54);
      in_dout(56) <= data_inter(62);
      in_dout(55) <= data_inter(4);
      in_dout(54) <= data_inter(12);
      in_dout(53) <= data_inter(20);
      in_dout(52) <= data_inter(28);
      in_dout(51) <= data_inter(36);
      in_dout(50) <= data_inter(44);
      in_dout(49) <= data_inter(52);
      in_dout(48) <= data_inter(60);
      in_dout(47) <= data_inter(2);
      in_dout(46) <= data_inter(10);
      in_dout(45) <= data_inter(18);
      in_dout(44) <= data_inter(26);
      in_dout(43) <= data_inter(34);
      in_dout(42) <= data_inter(42);
      in_dout(41) <= data_inter(50);
      in_dout(40) <= data_inter(58);
      in_dout(39) <= data_inter(0);
      in_dout(38) <= data_inter(8);
      in_dout(37) <= data_inter(16);
      in_dout(36) <= data_inter(24);
      in_dout(35) <= data_inter(32);
      in_dout(34) <= data_inter(40);
      in_dout(33) <= data_inter(48);
      in_dout(32) <= data_inter(56);
      in_dout(31) <= data_inter(7);
      in_dout(30) <= data_inter(15);
      in_dout(29) <= data_inter(23);
```

```
        in_dout(28) <= data_inter(31);
        in_dout(27) <= data_inter(39);
        in_dout(26) <= data_inter(47);
        in_dout(25) <= data_inter(55);
        in_dout(24) <= data_inter(63);
        in_dout(23) <= data_inter(5);
        in_dout(22) <= data_inter(13);
        in_dout(21) <= data_inter(21);
        in_dout(20) <= data_inter(29);
        in_dout(19) <= data_inter(37);
        in_dout(18) <= data_inter(45);
        in_dout(17) <= data_inter(53);
        in_dout(16) <= data_inter(61);
        in_dout(15) <= data_inter(3);
        in_dout(14) <= data_inter(11);
        in_dout(13) <= data_inter(19);
        in_dout(12) <= data_inter(27);
        in_dout(11) <= data_inter(35);
        in_dout(10) <= data_inter(43);
        in_dout(9)  <= data_inter(51);
        in_dout(8)  <= data_inter(59);
        in_dout(7)  <= data_inter(1);
        in_dout(6)  <= data_inter(9);
        in_dout(5)  <= data_inter(17);
        in_dout(4)  <= data_inter(25);
        in_dout(3)  <= data_inter(33);
        in_dout(2)  <= data_inter(41);
        in_dout(1)  <= data_inter(49);
        in_dout(0)  <= data_inter(57);
        end if;
        end process out_process;

END comport;

-------------------------------------------------------------------

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Reg_32 IS
        PORT( CLK_Reg     : in std_logic;
              Reg_in      : in std_logic_vector ( 31 downto 0 );
              Reg_out     : out std_logic_vector ( 31 downto 0 ));
END Reg_32;

ARCHITECTURE comport OF Reg_32 IS

BEGIN
        Reg_process : PROCESS (Reg_in, CLK_Reg)
        BEGIN
          if (CLK_Reg'event and CLK_Reg = '1') then
             Reg_out <= Reg_in;
          end if;
        END PROCESS Reg_process;
END comport;

-------------------------------------------------------------------

LIBRARY ieee ;
```



```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity FSM is
        port (CLK_cycle   : in  std_logic;
              start_FSM   : in std_logic;
              FSM_out     : out std_logic_vector ( 3 downto 0 ));
end FSM;

architecture state_machine of FSM is

TYPE FSM_state is (
state_1, state_2,  state_3,  state_4,  state_5,  state_6,  state_7,  state_8,
state_9, state_10, state_11, state_12, state_13, state_14, state_15, state_16);

signal current_state : FSM_state ;
signal next_state    : FSM_state ;

begin
   FSM_process : process
   begin

      wait until ( CLK_cycle' event and CLK_cycle ='1');
      if (start_FSM = '1') then
      case current_state is
         when state_1  =>
            FSM_out <= "0000";
            next_state <= state_2;
         when state_2  =>
            FSM_out <= "0001";
            next_state <= state_3;
         when state_3  =>
            FSM_out <= "0010";
            next_state <= state_4;
         when state_4  =>
            FSM_out <= "0011";
            next_state <= state_5;
         when state_5  =>
            FSM_out <= "0100";
            next_state <= state_6;
         when state_6  =>
            FSM_out <= "0101";
            next_state <= state_7;
         when state_7  =>
            FSM_out <= "0110";
            next_state <= state_8;
         when state_8  =>
            FSM_out <= "0111";
            next_state <= state_9;
         when state_9  =>
            FSM_out <= "1000";
            next_state <= state_10;
         when state_10  =>
            FSM_out <= "1001";
            next_state <= state_11;
         when state_11  =>
            FSM_out <= "1010";
            next_state <= state_12;
         when state_12  =>
            FSM_out <= "1011";
```

```
            next_state <= state_13;
         when state_13  =>
            FSM_out <= "1100";
            next_state <= state_14;
         when state_14  =>
            FSM_out <= "1101";
            next_state <= state_15;
         when state_15  =>
            FSM_out <= "1110";
            next_state <= state_16;
         when state_16  =>
            FSM_out <= "1111";
            next_state <= state_1;
        end case;
      else null;
      end if;
end process FSM_process;

SYNC: process
begin
        wait until ( CLK_cycle' event and CLK_cycle ='1');
        current_state <= next_state;
end process;
end state_machine;

-------------------------------------------------------------
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY MUX_2_1 is

        port (  start_M       : in std_logic;
                MUX_Reg_64    : in std_logic_vector ( 63 downto 0 );
                MUX_Reg_L     : in std_logic_vector ( 31 downto 0 );
                MUX_Reg_R     : in std_logic_vector ( 31 downto 0 );
                MUX_sel       : in std_logic_vector ( 3 downto 0 );
                MUX_out_L     : out std_logic_vector ( 31 downto 0 );
                MUX_out_R     : out std_logic_vector ( 31 downto 0 ));
end MUX_2_1;

architecture DATAFLOW OF MUX_2_1 is

begin
   MUX_process :process ( MUX_Reg_64, MUX_Reg_L, MUX_Reg_R, MUX_sel)
   begin
        if (start_M = '1') then
                case  MUX_sel is
                when "0000" =>
                        MUX_out_R(31) <= MUX_Reg_64(63);
                        MUX_out_R(30) <= MUX_Reg_64(62);
                        MUX_out_R(29) <= MUX_Reg_64(61);
                        MUX_out_R(28) <= MUX_Reg_64(60);
                        MUX_out_R(27) <= MUX_Reg_64(59);
                        MUX_out_R(26) <= MUX_Reg_64(58);
                        MUX_out_R(25) <= MUX_Reg_64(57);
                        MUX_out_R(24) <= MUX_Reg_64(56);
                        MUX_out_R(23) <= MUX_Reg_64(55);
                        MUX_out_R(22) <= MUX_Reg_64(54);
                        MUX_out_R(21) <= MUX_Reg_64(53);
```



```
                        MUX_out_R(20) <= MUX_Reg_64(52);
                        MUX_out_R(19) <= MUX_Reg_64(51);
                        MUX_out_R(18) <= MUX_Reg_64(50);
                        MUX_out_R(17) <= MUX_Reg_64(49);
                        MUX_out_R(16) <= MUX_Reg_64(48);
                        MUX_out_R(15) <= MUX_Reg_64(47);
                        MUX_out_R(14) <= MUX_Reg_64(46);
                        MUX_out_R(13) <= MUX_Reg_64(45);
                        MUX_out_R(12) <= MUX_Reg_64(44);
                        MUX_out_R(11) <= MUX_Reg_64(43);
                        MUX_out_R(10) <= MUX_Reg_64(42);
                        MUX_out_R(9) <= MUX_Reg_64(41);
                        MUX_out_R(8) <= MUX_Reg_64(40);
                        MUX_out_R(7) <= MUX_Reg_64(39);
                        MUX_out_R(6) <= MUX_Reg_64(38);
                        MUX_out_R(5) <= MUX_Reg_64(37);
                        MUX_out_R(4) <= MUX_Reg_64(36);
                        MUX_out_R(3) <= MUX_Reg_64(35);
                        MUX_out_R(2) <= MUX_Reg_64(34);
                        MUX_out_R(1) <= MUX_Reg_64(33);
                        MUX_out_R(0) <= MUX_Reg_64(32);

                        MUX_out_L(31) <= MUX_Reg_64(31);
                        MUX_out_L(30) <= MUX_Reg_64(30);
                        MUX_out_L(29) <= MUX_Reg_64(29);
                        MUX_out_L(28) <= MUX_Reg_64(28);
                        MUX_out_L(27) <= MUX_Reg_64(27);
                        MUX_out_L(26) <= MUX_Reg_64(26);
                        MUX_out_L(25) <= MUX_Reg_64(25);
                        MUX_out_L(24) <= MUX_Reg_64(24);
                        MUX_out_L(23) <= MUX_Reg_64(23);
                        MUX_out_L(22) <= MUX_Reg_64(22);
                        MUX_out_L(21) <= MUX_Reg_64(21);
                        MUX_out_L(20) <= MUX_Reg_64(20);
                        MUX_out_L(19) <= MUX_Reg_64(19);
                        MUX_out_L(18) <= MUX_Reg_64(18);
                        MUX_out_L(17) <= MUX_Reg_64(17);
                        MUX_out_L(16) <= MUX_Reg_64(16);
                        MUX_out_L(15) <= MUX_Reg_64(15);
                        MUX_out_L(14) <= MUX_Reg_64(14);
                        MUX_out_L(13) <= MUX_Reg_64(13);
                        MUX_out_L(12) <= MUX_Reg_64(12);
                        MUX_out_L(11) <= MUX_Reg_64(11);
                        MUX_out_L(10) <= MUX_Reg_64(10);
                        MUX_out_L(9) <= MUX_Reg_64(9);
                        MUX_out_L(8) <= MUX_Reg_64(8);
                        MUX_out_L(7) <= MUX_Reg_64(7);
                        MUX_out_L(6) <= MUX_Reg_64(6);
                        MUX_out_L(5) <= MUX_Reg_64(5);
                        MUX_out_L(4) <= MUX_Reg_64(4);
                        MUX_out_L(3) <= MUX_Reg_64(3);
                        MUX_out_L(2) <= MUX_Reg_64(2);
                        MUX_out_L(1) <= MUX_Reg_64(1);
                        MUX_out_L(0) <= MUX_Reg_64(0);

                when "0001" =>
                        MUX_out_L <= MUX_Reg_L;
                        MUX_out_R <= MUX_Reg_R;
                when "0010" =>
                        MUX_out_L <= MUX_Reg_L;
```

```
                MUX_out_R <= MUX_Reg_R;
            when "0011" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "0100" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "0101" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "0110" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "0111" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1000" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1001" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1010" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1011" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1100" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1101" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when "1110" =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            when  others =>
                MUX_out_L <= MUX_Reg_L;
                MUX_out_R <= MUX_Reg_R;
            end case;
        else null;
        end if;
        end process MUX_process;
end DATAFLOW;

-------------------------------------------------------------------

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY MUX_16_1 is

        port (  MUX_1   : in std_logic_vector ( 47 downto 0 );
                MUX_2   : in std_logic_vector ( 47 downto 0 );
                MUX_3   : in std_logic_vector ( 47 downto 0 );
                MUX_4   : in std_logic_vector ( 47 downto 0 );
                MUX_5   : in std_logic_vector ( 47 downto 0 );
                MUX_6   : in std_logic_vector ( 47 downto 0 );
```



```
                MUX_7   : in std_logic_vector ( 47 downto 0 );
                MUX_8   : in std_logic_vector ( 47 downto 0 );
                MUX_9   : in std_logic_vector ( 47 downto 0 );
                MUX_10  : in std_logic_vector ( 47 downto 0 );
                MUX_11  : in std_logic_vector ( 47 downto 0 );
                MUX_12  : in std_logic_vector ( 47 downto 0 );
                MUX_13  : in std_logic_vector ( 47 downto 0 );
                MUX_14  : in std_logic_vector ( 47 downto 0 );
                MUX_15  : in std_logic_vector ( 47 downto 0 );
                MUX_16  : in std_logic_vector ( 47 downto 0 );
                MUX_sel : in std_logic_vector ( 3 downto 0 );
                MUX_out : out std_logic_vector ( 47 downto 0 ));
end MUX_16_1;

architecture DATAFLOW OF MUX_16_1 is

begin
   MUX_process :process (MUX_1,  MUX_2,  MUX_3,  MUX_3,  MUX_4,  MUX_5,  MUX_6,
                         MUX_7,  MUX_8,  MUX_9,  MUX_10, MUX_11, MUX_12, MUX_13
                         MUX_14, MUX_15, MUX_16, MUX_sel)
          begin
                if ( MUX_sel = "0000") then
                        MUX_out <= MUX_1;
                elsif (MUX_sel = "0001") then
                        MUX_out <= MUX_2;
                elsif (MUX_sel = "0010") then
                        MUX_out <= MUX_3;
                elsif (MUX_sel = "0011") then
                        MUX_out <= MUX_4;
                elsif (MUX_sel = "0100") then
                        MUX_out <= MUX_5;
                elsif (MUX_sel = "0101") then
                        MUX_out <= MUX_6;
                elsif (MUX_sel = "0110") then
                        MUX_out <= MUX_7;
                elsif (MUX_sel = "0111") then
                        MUX_out <= MUX_8;
                elsif (MUX_sel = "1000") then
                        MUX_out <= MUX_9;
                elsif (MUX_sel = "1001") then
                        MUX_out <= MUX_10;
                elsif (MUX_sel = "1010") then
                        MUX_out <= MUX_11;
                elsif (MUX_sel = "1011") then
                        MUX_out <= MUX_12;
                elsif (MUX_sel = "1100") then
                        MUX_out <= MUX_13;
                elsif (MUX_sel = "1101") then
                        MUX_out <= MUX_14;
                elsif (MUX_sel = "1110") then
                        MUX_out <= MUX_15;
                elsif (MUX_sel = "1111") then
                        MUX_out <= MUX_16;
                end if;
        end process MUX_process;
end DATAFLOW;

-------------------------------------------------------------------

LIBRARY ieee ;
```

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity KKK is
        port ( K_data    : in  std_logic_vector ( 63 downto 0);
               K1        : out std_logic_vector ( 47 downto 0 );
               K2        : out std_logic_vector ( 47 downto 0 );
               K3        : out std_logic_vector ( 47 downto 0 );
               K4        : out std_logic_vector ( 47 downto 0 );
               K5        : out std_logic_vector ( 47 downto 0 );
               K6        : out std_logic_vector ( 47 downto 0 );
               K7        : out std_logic_vector ( 47 downto 0 );
               K8        : out std_logic_vector ( 47 downto 0 );
               K9        : out std_logic_vector ( 47 downto 0 );
               K10       : out std_logic_vector ( 47 downto 0 );
               K11       : out std_logic_vector ( 47 downto 0 );
               K12       : out std_logic_vector ( 47 downto 0 );
               K13       : out std_logic_vector ( 47 downto 0 );
               K14       : out std_logic_vector ( 47 downto 0 );
               K15       : out std_logic_vector ( 47 downto 0 );
               K16       : out std_logic_vector ( 47 downto 0 ));
end KKK;

architecture COMPORT of KKK is

begin
   KEY1_process : process (K_data)
      begin

          K1(47)   <= K_data(54);
          K1(46)   <= K_data(13);
          K1(45)   <= K_data(30);
          K1(44)   <= K_data(4);
          K1(43)   <= K_data(15);
          K1(42)   <= K_data(47);
          K1(41)   <= K_data(31);
          K1(40)   <= K_data(7);
          K1(39)   <= K_data(62);
          K1(38)   <= K_data(55);
          K1(37)   <= K_data(45);
          K1(36)   <= K_data(22);
          K1(35)   <= K_data(61);
          K1(34)   <= K_data(29);
          K1(33)   <= K_data(38);
          K1(32)   <= K_data(39);
          K1(31)   <= K_data(20);
          K1(30)   <= K_data(6);
          K1(29)   <= K_data(5);
          K1(28)   <= K_data(63);
          K1(27)   <= K_data(28);
          K1(26)   <= K_data(37);
          K1(25)   <= K_data(46);
          K1(24)   <= K_data(23);
          K1(23)   <= K_data(42);
          K1(22)   <= K_data(36);
          K1(21)   <= K_data(25);
          K1(20)   <= K_data(9);
          K1(19)   <= K_data(27);
          K1(18)   <= K_data(60);
          K1(17)   <= K_data(17);
```



```
          K1(16)   <= K_data(34);
          K1(15)   <= K_data(59);
          K1(14)   <= K_data(11);
          K1(13)   <= K_data(41);
          K1(12)   <= K_data(35);
          K1(11)   <= K_data(3);
          K1(10)   <= K_data(43);
          K1(9)    <= K_data(26);
          K1(8)    <= K_data(1);
          K1(7)    <= K_data(49);
          K1(6)    <= K_data(44);
          K1(5)    <= K_data(19);
          K1(4)    <= K_data(50);
          K1(3)    <= K_data(51);
          K1(2)    <= K_data(2);
          K1(1)    <= K_data(9);
          K1(0)    <= K_data(33);

   end process KEY1_process;

KEY2_process : process (K_data)
      begin

          K2(47)   <= K_data(62);
          K2(46)   <= K_data(21);
          K2(45)   <= K_data(38);
          K2(44)   <= K_data(12);
          K2(43)   <= K_data(23);
          K2(42)   <= K_data(55);
          K2(41)   <= K_data(39);
          K2(40)   <= K_data(15);
          K2(39)   <= K_data(5);
          K2(38)   <= K_data(63);
          K2(37)   <= K_data(53);
          K2(36)   <= K_data(30);
          K2(35)   <= K_data(4);
          K2(34)   <= K_data(37);
          K2(33)   <= K_data(46);
          K2(32)   <= K_data(47);
          K2(31)   <= K_data(28);
          K2(30)   <= K_data(14);
          K2(29)   <= K_data(13);
          K2(28)   <= K_data(6);
          K2(27)   <= K_data(7);
          K2(26)   <= K_data(45);
          K2(25)   <= K_data(54);
          K2(24)   <= K_data(31);
          K2(23)   <= K_data(50);
          K2(22)   <= K_data(44);
          K2(21)   <= K_data(33);
          K2(20)   <= K_data(15);
          K2(19)   <= K_data(35);
          K2(18)   <= K_data(1);
          K2(17)   <= K_data(25);
          K2(16)   <= K_data(42);
          K2(15)   <= K_data(36);
          K2(14)   <= K_data(19);
          K2(13)   <= K_data(49);
          K2(12)   <= K_data(43);
          K2(11)   <= K_data(11);
```

```
        K2(10)    <= K_data(51);
        K2(9)     <= K_data(34);
        K2(8)     <= K_data(9);
        K2(7)     <= K_data(57);
        K2(6)     <= K_data(52);
        K2(5)     <= K_data(27);
        K2(4)     <= K_data(58);
        K2(3)     <= K_data(59);
        K2(2)     <= K_data(9);
        K2(1)     <= K_data(17);
        K2(0)     <= K_data(41);

    end process KEY2_process;

KEY3_process : process (K_data)
      begin

        K3(47)    <= K_data(13);
        K3(46)    <= K_data(37);
        K3(45)    <= K_data(54);
        K3(44)    <= K_data(28);
        K3(43)    <= K_data(39);
        K3(42)    <= K_data(6);
        K3(41)    <= K_data(55);
        K3(40)    <= K_data(31);
        K3(39)    <= K_data(21);
        K3(38)    <= K_data(14);
        K3(37)    <= K_data(4);
        K3(36)    <= K_data(46);
        K3(35)    <= K_data(20);
        K3(34)    <= K_data(53);
        K3(33)    <= K_data(62);
        K3(32)    <= K_data(63);
        K3(31)    <= K_data(15);
        K3(30)    <= K_data(30);
        K3(29)    <= K_data(29);
        K3(28)    <= K_data(22);
        K3(27)    <= K_data(23);
        K3(26)    <= K_data(61);
        K3(25)    <= K_data(5);
        K3(24)    <= K_data(47);
        K3(23)    <= K_data(3);
        K3(22)    <= K_data(60);
        K3(21)    <= K_data(49);
        K3(20)    <= K_data(34);
        K3(19)    <= K_data(51);
        K3(18)    <= K_data(17);
        K3(17)    <= K_data(41);
        K3(16)    <= K_data(58);
        K3(15)    <= K_data(52);
        K3(14)    <= K_data(35);
        K3(13)    <= K_data(2);
        K3(12)    <= K_data(59);
        K3(11)    <= K_data(27);
        K3(10)    <= K_data(36);
        K3(9)     <= K_data(50);
        K3(8)     <= K_data(25);
        K3(7)     <= K_data(9);
        K3(6)     <= K_data(1);
        K3(5)     <= K_data(43);
```



```
        K3(4)     <= K_data(11);
        K3(3)     <= K_data(44);
        K3(2)     <= K_data(26);
        K3(1)     <= K_data(33);
        K3(0)     <= K_data(57);

    end process KEY3_process;

KEY4_process : process (K_data)
      begin

        K4(47)    <= K_data(29);
        K4(46)    <= K_data(53);
        K4(45)    <= K_data(5);
        K4(44)    <= K_data(15);
        K4(43)    <= K_data(55);
        K4(42)    <= K_data(22);
        K4(41)    <= K_data(6);
        K4(40)    <= K_data(47);
        K4(39)    <= K_data(37);
        K4(38)    <= K_data(30);
        K4(37)    <= K_data(20);
        K4(36)    <= K_data(62);
        K4(35)    <= K_data(7);
        K4(34)    <= K_data(4);
        K4(33)    <= K_data(13);
        K4(32)    <= K_data(14);
        K4(31)    <= K_data(31);
        K4(30)    <= K_data(46);
        K4(29)    <= K_data(45);
        K4(28)    <= K_data(38);
        K4(27)    <= K_data(39);
        K4(26)    <= K_data(12);
        K4(25)    <= K_data(21);
        K4(24)    <= K_data(63);
        K4(23)    <= K_data(19);
        K4(22)    <= K_data(9);
        K4(21)    <= K_data(2);
        K4(20)    <= K_data(50);
        K4(19)    <= K_data(36);
        K4(18)    <= K_data(33);
        K4(17)    <= K_data(57);
        K4(16)    <= K_data(11);
        K4(15)    <= K_data(1);
        K4(14)    <= K_data(51);
        K4(13)    <= K_data(18);
        K4(12)    <= K_data(44);
        K4(11)    <= K_data(43);
        K4(10)    <= K_data(52);
        K4(9)     <= K_data(3);
        K4(8)     <= K_data(41);
        K4(7)     <= K_data(25);
        K4(6)     <= K_data(17);
        K4(5)     <= K_data(59);
        K4(4)     <= K_data(27);
        K4(3)     <= K_data(60);
        K4(2)     <= K_data(42);
        K4(1)     <= K_data(49);
        K4(0)     <= K_data(9);
```

```
    end process KEY4_process;

KEY5_process : process (K_data)
      begin

          K5(47)    <= K_data(45);
          K5(46)    <= K_data(4);
          K5(45)    <= K_data(21);
          K5(44)    <= K_data(31);
          K5(43)    <= K_data(6);
          K5(42)    <= K_data(38);
          K5(41)    <= K_data(22);
          K5(40)    <= K_data(63);
          K5(39)    <= K_data(53);
          K5(38)    <= K_data(46);
          K5(37)    <= K_data(7);
          K5(36)    <= K_data(13);
          K5(35)    <= K_data(23);
          K5(34)    <= K_data(20);
          K5(33)    <= K_data(29);
          K5(32)    <= K_data(30);
          K5(31)    <= K_data(47);
          K5(30)    <= K_data(62);
          K5(29)    <= K_data(61);
          K5(28)    <= K_data(54);
          K5(27)    <= K_data(55);
          K5(26)    <= K_data(28);
          K5(25)    <= K_data(37);
          K5(24)    <= K_data(14);
          K5(23)    <= K_data(25);
          K5(22)    <= K_data(25);
          K5(21)    <= K_data(18);
          K5(20)    <= K_data(3);
          K5(19)    <= K_data(52);
          K5(18)    <= K_data(49);
          K5(17)    <= K_data(9);
          K5(16)    <= K_data(27);
          K5(15)    <= K_data(17);
          K5(14)    <= K_data(36);
          K5(13)    <= K_data(34);
          K5(12)    <= K_data(60);
          K5(11)    <= K_data(59);
          K5(10)    <= K_data(1);
          K5(9)     <= K_data(19);
          K5(8)     <= K_data(57);
          K5(7)     <= K_data(42);
          K5(6)     <= K_data(33);
          K5(5)     <= K_data(44);
          K5(4)     <= K_data(43);
          K5(3)     <= K_data(9);
          K5(2)     <= K_data(58);
          K5(1)     <= K_data(2);
          K5(0)      <= K_data(26);

   end process KEY5_process;

   KEY6_process : process (K_data)
      begin
```



```
          K6(47)    <= K_data(61);
          K6(46)    <= K_data(20);
          K6(45)    <= K_data(37);
          K6(44)    <= K_data(47);
          K6(43)    <= K_data(22);
          K6(42)    <= K_data(54);
          K6(41)    <= K_data(38);
          K6(40)    <= K_data(14);
          K6(39)    <= K_data(4);
          K6(38)    <= K_data(62);
          K6(37)    <= K_data(23);
          K6(36)    <= K_data(29);
          K6(35)    <= K_data(39);
          K6(34)    <= K_data(7);
          K6(33)    <= K_data(45);
          K6(32)    <= K_data(46);
          K6(31)    <= K_data(63);
          K6(30)    <= K_data(13);
          K6(29)    <= K_data(12);
          K6(28)    <= K_data(5);
          K6(27)    <= K_data(6);
          K6(26)    <= K_data(15);
          K6(25)    <= K_data(53);
          K6(24)    <= K_data(30);
          K6(23)    <= K_data(51);
          K6(22)    <= K_data(41);
          K6(21)    <= K_data(34);
          K6(20)    <= K_data(19);
          K6(19)    <= K_data(1);
          K6(18)    <= K_data(2);
          K6(17)    <= K_data(26);
          K6(16)    <= K_data(43);
          K6(15)    <= K_data(33);
          K6(14)    <= K_data(52);
          K6(13)    <= K_data(50);
          K6(12)    <= K_data(9);
          K6(11)    <= K_data(44);
          K6(10)    <= K_data(17);
          K6(9)     <= K_data(35);
          K6(8)     <= K_data(9);
          K6(7)     <= K_data(58);
          K6(6)     <= K_data(49);
          K6(5)     <= K_data(60);
          K6(4)     <= K_data(59);
          K6(3)     <= K_data(25);
          K6(2)     <= K_data(11);
          K6(1)     <= K_data(18);
          K6(0)     <= K_data(42);

   end process KEY6_process;

KEY7_process : process (K_data)
      begin

          K7(47)    <= K_data(12);
          K7(46)    <= K_data(7);
          K7(45)    <= K_data(53);
          K7(44)    <= K_data(63);
          K7(43)    <= K_data(38);
          K7(42)    <= K_data(5);
```

```
            K7(41)   <= K_data(54);
            K7(40)   <= K_data(30);
            K7(39)   <= K_data(20);
            K7(38)   <= K_data(13);
            K7(37)   <= K_data(39);
            K7(36)   <= K_data(45);
            K7(35)   <= K_data(55);
            K7(34)   <= K_data(23);
            K7(33)   <= K_data(61);
            K7(32)   <= K_data(62);
            K7(31)   <= K_data(14);
            K7(30)   <= K_data(29);
            K7(29)   <= K_data(28);
            K7(28)   <= K_data(21);
            K7(27)   <= K_data(22);
            K7(26)   <= K_data(31);
            K7(25)   <= K_data(4);
            K7(24)   <= K_data(46);
            K7(23)   <= K_data(36);
            K7(22)   <= K_data(57);
            K7(21)   <= K_data(50);
            K7(20)   <= K_data(35);
            K7(19)   <= K_data(17);
            K7(18)   <= K_data(18);
            K7(17)   <= K_data(42);
            K7(16)   <= K_data(59);
            K7(15)   <= K_data(49);
            K7(14)   <= K_data(1);
            K7(13)   <= K_data(3);
            K7(12)   <= K_data(25);
            K7(11)   <= K_data(60);
            K7(10)   <= K_data(33);
            K7(9)    <= K_data(51);
            K7(8)    <= K_data(26);
            K7(7)    <= K_data(11);
            K7(6)    <= K_data(2);
            K7(5)    <= K_data(9);
            K7(4)    <= K_data(44);
            K7(3)    <= K_data(41);
            K7(2)    <= K_data(27);
            K7(1)    <= K_data(34);
            K7(0)    <= K_data(58);

    end process KEY7_process;

KEY8_process : process (K_data)
      begin

            K8(47)   <= K_data(28);
            K8(46)   <= K_data(23);
            K8(45)   <= K_data(4);
            K8(44)   <= K_data(14);
            K8(43)   <= K_data(54);
            K8(42)   <= K_data(21);
            K8(41)   <= K_data(5);
            K8(40)   <= K_data(46);
            K8(39)   <= K_data(7);
            K8(38)   <= K_data(29);
            K8(37)   <= K_data(55);
            K8(36)   <= K_data(61);
```



```
            K8(35)   <= K_data(6);
            K8(34)   <= K_data(39);
            K8(33)   <= K_data(12);
            K8(32)   <= K_data(13);
            K8(31)   <= K_data(30);
            K8(30)   <= K_data(45);
            K8(29)   <= K_data(15);
            K8(28)   <= K_data(37);
            K8(27)   <= K_data(38);
            K8(26)   <= K_data(47);
            K8(25)   <= K_data(20);
            K8(24)   <= K_data(62);
            K8(23)   <= K_data(52);
            K8(22)   <= K_data(9);
            K8(21)   <= K_data(3);
            K8(20)   <= K_data(51);
            K8(19)   <= K_data(33);
            K8(18)   <= K_data(43);
            K8(17)   <= K_data(58);
            K8(16)   <= K_data(44);
            K8(15)   <= K_data(2);
            K8(14)   <= K_data(17);
            K8(13)   <= K_data(19);
            K8(12)   <= K_data(41);
            K8(11)   <= K_data(9);
            K8(10)   <= K_data(49);
            K8(9)    <= K_data(36);
            K8(8)    <= K_data(42);
            K8(7)    <= K_data(27);
            K8(6)    <= K_data(18);
            K8(5)    <= K_data(25);
            K8(4)    <= K_data(60);
            K8(3)    <= K_data(57);
            K8(2)    <= K_data(43);
            K8(1)    <= K_data(50);
            K8(0)    <= K_data(11);

    end process KEY8_process;

KEY9_process : process (K_data)
      begin

            K9(47)   <= K_data(7);
            K9(46)   <= K_data(31);
            K9(45)   <= K_data(12);
            K9(44)   <= K_data(22);
            K9(43)   <= K_data(62);
            K9(42)   <= K_data(29);
            K9(41)   <= K_data(13);
            K9(40)   <= K_data(54);
            K9(39)   <= K_data(15);
            K9(38)   <= K_data(37);
            K9(37)   <= K_data(63);
            K9(36)   <= K_data(4);
            K9(35)   <= K_data(14);
            K9(34)   <= K_data(47);
            K9(33)   <= K_data(20);
            K9(32)   <= K_data(21);
            K9(31)   <= K_data(38);
            K9(30)   <= K_data(53);
```

```
        K9(29)   <= K_data(23);
        K9(28)   <= K_data(45);
        K9(27)   <= K_data(46);
        K9(26)   <= K_data(55);
        K9(25)   <= K_data(28);
        K9(24)   <= K_data(5);
        K9(23)   <= K_data(60);
        K9(22)   <= K_data(18);
        K9(21)   <= K_data(11);
        K9(20)   <= K_data(59);
        K9(19)   <= K_data(41);
        K9(18)   <= K_data(42);
        K9(17)   <= K_data(3);
        K9(16)   <= K_data(52);
        K9(15)   <= K_data(9);
        K9(14)   <= K_data(25);
        K9(13)   <= K_data(27);
        K9(12)   <= K_data(49);
        K9(11)   <= K_data(17);
        K9(10)   <= K_data(57);
        K9(9)    <= K_data(44);
        K9(8)    <= K_data(50);
        K9(7)    <= K_data(35);
        K9(6)    <= K_data(26);
        K9(5)    <= K_data(33);
        K9(4)    <= K_data(1);
        K9(3)    <= K_data(2);
        K9(2)    <= K_data(51);
        K9(1)    <= K_data(58);
        K9(0)    <= K_data(19);

    end process KEY9_process;


KEY10_process : process (K_data)
    begin

        K10(47)   <= K_data(23);
        K10(46)   <= K_data(47);
        K10(45)   <= K_data(28);
        K10(44)   <= K_data(38);
        K10(43)   <= K_data(13);
        K10(42)   <= K_data(45);
        K10(41)   <= K_data(29);
        K10(40)   <= K_data(5);
        K10(39)   <= K_data(31);
        K10(38)   <= K_data(53);
        K10(37)   <= K_data(14);
        K10(36)   <= K_data(20);
        K10(35)   <= K_data(30);
        K10(34)   <= K_data(63);
        K10(33)   <= K_data(7);
        K10(32)   <= K_data(37);
        K10(31)   <= K_data(54);
        K10(30)   <= K_data(4);
        K10(29)   <= K_data(39);
        K10(28)   <= K_data(61);
        K10(27)   <= K_data(62);
        K10(26)   <= K_data(6);
        K10(25)   <= K_data(15);
```



```
        K10(24)   <= K_data(21);
        K10(23)   <= K_data(9);
        K10(22)   <= K_data(34);
        K10(21)   <= K_data(27);
        K10(20)   <= K_data(44);
        K10(19)   <= K_data(57);
        K10(18)   <= K_data(58);
        K10(17)   <= K_data(19);
        K10(16)   <= K_data(1);
        K10(15)   <= K_data(26);
        K10(14)   <= K_data(41);
        K10(13)   <= K_data(43);
        K10(12)   <= K_data(2);
        K10(11)   <= K_data(33);
        K10(10)   <= K_data(9);
        K10(9)    <= K_data(60);
        K10(8)    <= K_data(3);
        K10(7)    <= K_data(51);
        K10(6)    <= K_data(42);
        K10(5)    <= K_data(49);
        K10(4)    <= K_data(17);
        K10(3)    <= K_data(18);
        K10(2)    <= K_data(36);
        K10(1)    <= K_data(11);
        K10(0)    <= K_data(35);

end process KEY10_process;


KEY11_process : process (K_data)
   begin

        K11(47)   <= K_data(39);
        K11(46)   <= K_data(63);
        K11(45)   <= K_data(15);
        K11(44)   <= K_data(54);
        K11(43)   <= K_data(29);
        K11(42)   <= K_data(61);
        K11(41)   <= K_data(45);
        K11(40)   <= K_data(21);
        K11(39)   <= K_data(47);
        K11(38)   <= K_data(4);
        K11(37)   <= K_data(30);
        K11(36)   <= K_data(7);
        K11(35)   <= K_data(46);
        K11(34)   <= K_data(14);
        K11(33)   <= K_data(23);
        K11(32)   <= K_data(53);
        K11(31)   <= K_data(5);
        K11(30)   <= K_data(20);
        K11(29)   <= K_data(55);
        K11(28)   <= K_data(12);
        K11(27)   <= K_data(13);
        K11(26)   <= K_data(22);
        K11(25)   <= K_data(31);
        K11(24)   <= K_data(37);
        K11(23)   <= K_data(25);
        K11(22)   <= K_data(50);
        K11(21)   <= K_data(43);
        K11(20)   <= K_data(60);
```

```
        K11(19)   <= K_data(9);
        K11(18)   <= K_data(11);
        K11(17)   <= K_data(35);
        K11(16)   <= K_data(17);
        K11(15)   <= K_data(42);
        K11(14)   <= K_data(57);
        K11(13)   <= K_data(59);
        K11(12)   <= K_data(18);
        K11(11)   <= K_data(49);
        K11(10)   <= K_data(26);
        K11(9)    <= K_data(9);
        K11(8)    <= K_data(19);
        K11(7)    <= K_data(36);
        K11(6)    <= K_data(58);
        K11(5)    <= K_data(2);
        K11(4)    <= K_data(33);
        K11(3)    <= K_data(34);
        K11(2)    <= K_data(52);
        K11(1)    <= K_data(27);
        K11(0)    <= K_data(51);

end process KEY11_process;


KEY12_process : process (K_data)
   begin

        K12(47)   <= K_data(55);
        K12(46)   <= K_data(14);
        K12(45)   <= K_data(31);
        K12(44)   <= K_data(5);
        K12(43)   <= K_data(45);
        K12(42)   <= K_data(12);
        K12(41)   <= K_data(61);
        K12(40)   <= K_data(37);
        K12(39)   <= K_data(63);
        K12(38)   <= K_data(20);
        K12(37)   <= K_data(46);
        K12(36)   <= K_data(23);
        K12(35)   <= K_data(62);
        K12(34)   <= K_data(30);
        K12(33)   <= K_data(39);
        K12(32)   <= K_data(4);
        K12(31)   <= K_data(21);
        K12(30)   <= K_data(7);
        K12(29)   <= K_data(6);
        K12(28)   <= K_data(28);
        K12(27)   <= K_data(29);
        K12(26)   <= K_data(38);
        K12(25)   <= K_data(47);
        K12(24)   <= K_data(53);
        K12(23)   <= K_data(41);
        K12(22)   <= K_data(3);
        K12(21)   <= K_data(59);
        K12(20)   <= K_data(9);
        K12(19)   <= K_data(26);
        K12(18)   <= K_data(27);
        K12(17)   <= K_data(51);
        K12(16)   <= K_data(33);
        K12(15)   <= K_data(58);
```



```
        K12(14)   <= K_data(9);
        K12(13)   <= K_data(44);
        K12(12)   <= K_data(34);
        K12(11)   <= K_data(2);
        K12(10)   <= K_data(42);
        K12(9)    <= K_data(25);
        K12(8)    <= K_data(35);
        K12(7)    <= K_data(52);
        K12(6)    <= K_data(11);
        K12(5)    <= K_data(18);
        K12(4)    <= K_data(49);
        K12(3)    <= K_data(50);
        K12(2)    <= K_data(1);
        K12(1)    <= K_data(43);
        K12(0)    <= K_data(36);

end process KEY12_process;


KEY13_process : process (K_data)
   begin

        K13(47)   <= K_data(6);
        K13(46)   <= K_data(30);
        K13(45)   <= K_data(47);
        K13(44)   <= K_data(21);
        K13(43)   <= K_data(61);
        K13(42)   <= K_data(28);
        K13(41)   <= K_data(12);
        K13(40)   <= K_data(53);
        K13(39)   <= K_data(14);
        K13(38)   <= K_data(7);
        K13(37)   <= K_data(62);
        K13(36)   <= K_data(39);
        K13(35)   <= K_data(13);
        K13(34)   <= K_data(46);
        K13(33)   <= K_data(55);
        K13(32)   <= K_data(20);
        K13(31)   <= K_data(37);
        K13(30)   <= K_data(23);
        K13(29)   <= K_data(22);
        K13(28)   <= K_data(15);
        K13(27)   <= K_data(45);
        K13(26)   <= K_data(54);
        K13(25)   <= K_data(63);
        K13(24)   <= K_data(4);
        K13(23)   <= K_data(57);
        K13(22)   <= K_data(19);
        K13(21)   <= K_data(44);
        K13(20)   <= K_data(25);
        K13(19)   <= K_data(42);
        K13(18)   <= K_data(43);
        K13(17)   <= K_data(36);
        K13(16)   <= K_data(49);
        K13(15)   <= K_data(11);
        K13(14)   <= K_data(26);
        K13(13)   <= K_data(60);
        K13(12)   <= K_data(50);
        K13(11)   <= K_data(18);
        K13(10)   <= K_data(58);
```

```
        K13(9)    <= K_data(41);
        K13(8)    <= K_data(51);
        K13(7)    <= K_data(1);
        K13(6)    <= K_data(27);
        K13(5)    <= K_data(34);
        K13(4)    <= K_data(2);
        K13(3)    <= K_data(3);
        K13(2)    <= K_data(17);
        K13(1)    <= K_data(59);
        K13(0)    <= K_data(52);

  end process KEY13_process;


KEY14_process : process (K_data)
    begin

        K14(47)   <= K_data(22);
        K14(46)   <= K_data(46);
        K14(45)   <= K_data(63);
        K14(44)   <= K_data(37);
        K14(43)   <= K_data(12);
        K14(42)   <= K_data(15);
        K14(41)   <= K_data(28);
        K14(40)   <= K_data(4);
        K14(39)   <= K_data(30);
        K14(38)   <= K_data(23);
        K14(37)   <= K_data(13);
        K14(36)   <= K_data(55);
        K14(35)   <= K_data(29);
        K14(34)   <= K_data(62);
        K14(33)   <= K_data(6);
        K14(32)   <= K_data(7);
        K14(31)   <= K_data(53);
        K14(30)   <= K_data(39);
        K14(29)   <= K_data(38);
        K14(28)   <= K_data(31);
        K14(27)   <= K_data(61);
        K14(26)   <= K_data(5);
        K14(25)   <= K_data(14);
        K14(24)   <= K_data(20);
        K14(23)   <= K_data(9);
        K14(22)   <= K_data(35);
        K14(21)   <= K_data(60);
        K14(20)   <= K_data(41);
        K14(19)   <= K_data(58);
        K14(18)   <= K_data(59);
        K14(17)   <= K_data(52);
        K14(16)   <= K_data(2);
        K14(15)   <= K_data(27);
        K14(14)   <= K_data(42);
        K14(13)   <= K_data(9);
        K14(12)   <= K_data(3);
        K14(11)   <= K_data(34);
        K14(10)   <= K_data(11);
        K14(9)    <= K_data(57);
        K14(8)    <= K_data(36);
        K14(7)    <= K_data(17);
        K14(6)    <= K_data(43);
        K14(5)    <= K_data(50);
```



```
        K14(4)    <= K_data(18);
        K14(3)    <= K_data(19);
        K14(2)    <= K_data(34);
        K14(1)    <= K_data(44);
        K14(0)    <= K_data(1);

  end process KEY14_process;


KEY15_process : process (K_data)
   begin

        K15(47)   <= K_data(38);
        K15(46)   <= K_data(62);
        K15(45)   <= K_data(14);
        K15(44)   <= K_data(53);
        K15(43)   <= K_data(28);
        K15(42)   <= K_data(31);
        K15(41)   <= K_data(15);
        K15(40)   <= K_data(20);
        K15(39)   <= K_data(46);
        K15(38)   <= K_data(39);
        K15(37)   <= K_data(29);
        K15(36)   <= K_data(6);
        K15(35)   <= K_data(45);
        K15(34)   <= K_data(13);
        K15(33)   <= K_data(22);
        K15(32)   <= K_data(23);
        K15(31)   <= K_data(4);
        K15(30)   <= K_data(55);
        K15(29)   <= K_data(54);
        K15(28)   <= K_data(47);
        K15(27)   <= K_data(12);
        K15(26)   <= K_data(21);
        K15(25)   <= K_data(30);
        K15(24)   <= K_data(7);
        K15(23)   <= K_data(26);
        K15(22)   <= K_data(51);
        K15(21)   <= K_data(9);
        K15(20)   <= K_data(57);
        K15(19)   <= K_data(11);
        K15(18)   <= K_data(44);
        K15(17)   <= K_data(1);
        K15(16)   <= K_data(18);
        K15(15)   <= K_data(43);
        K15(14)   <= K_data(58);
        K15(13)   <= K_data(25);
        K15(12)   <= K_data(19);
        K15(11)   <= K_data(50);
        K15(10)   <= K_data(27);
        K15(9)    <= K_data(9);
        K15(8)    <= K_data(52);
        K15(7)    <= K_data(33);
        K15(6)    <= K_data(59);
        K15(5)    <= K_data(3);
        K15(4)    <= K_data(34);
        K15(3)    <= K_data(35);
        K15(2)    <= K_data(49);
        K15(1)    <= K_data(60);
        K15(0)    <= K_data(17);
```

```
    end process KEY15_process;

    KEY16_process : process (K_data)
       begin

            K16(47)   <= K_data(46);
            K16(46)   <= K_data(5);
            K16(45)   <= K_data(22);
            K16(44)   <= K_data(61);
            K16(43)   <= K_data(7);
            K16(42)   <= K_data(39);
            K16(41)   <= K_data(23);
            K16(40)   <= K_data(28);
            K16(39)   <= K_data(54);
            K16(38)   <= K_data(47);
            K16(37)   <= K_data(37);
            K16(36)   <= K_data(14);
            K16(35)   <= K_data(53);
            K16(34)   <= K_data(21);
            K16(33)   <= K_data(30);
            K16(32)   <= K_data(31);
            K16(31)   <= K_data(12);
            K16(30)   <= K_data(63);
            K16(29)   <= K_data(62);
            K16(28)   <= K_data(55);
            K16(27)   <= K_data(20);
            K16(26)   <= K_data(29);
            K16(25)   <= K_data(38);
            K16(24)   <= K_data(15);
            K16(23)   <= K_data(34);
            K16(22)   <= K_data(59);
            K16(21)   <= K_data(17);
            K16(20)   <= K_data(2);
            K16(19)   <= K_data(19);
            K16(18)   <= K_data(52);
            K16(17)   <= K_data(9);
            K16(16)   <= K_data(26);
            K16(15)   <= K_data(51);
            K16(14)   <= K_data(3);
            K16(13)   <= K_data(33);
            K16(12)   <= K_data(27);
            K16(11)   <= K_data(58);
            K16(10)   <= K_data(35);
            K16(9)    <= K_data(18);
            K16(8)    <= K_data(60);
            K16(7)    <= K_data(41);
            K16(6)    <= K_data(36);
            K16(5)    <= K_data(11);
            K16(4)    <= K_data(42);
            K16(3)    <= K_data(43);
            K16(2)    <= K_data(57);
            K16(1)    <= K_data(1);
            K16(0)    <= K_data(25);

    end process KEY16_process;

end COMPORT;
```



```
------------------------------------------------------------


LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity R_Reg is
        port( R_in     : in std_logic_vector ( 31 downto 0 );
              R_out    : out std_logic_vector ( 47 downto 0 );
              R_out_32 : out std_logic_vector ( 31 downto 0 ));
end R_Reg;

architecture comport of R_Reg is

begin
      Reg_R_process : process (R_in)
      begin
            R_out_32 <= R_in;
      end process Reg_R_process;

        R_out(47)  <= R_in(0);
        R_out(46)  <= R_in(31);
        R_out(45)  <= R_in(30);
        R_out(44)  <= R_in(29);
        R_out(43)  <= R_in(28);
        R_out(42)  <= R_in(27);
        R_out(41)  <= R_in(28);
        R_out(40)  <= R_in(27);
        R_out(39)  <= R_in(26);
        R_out(38)  <= R_in(25);
        R_out(37)  <= R_in(24);
        R_out(36)  <= R_in(23);
        R_out(35)  <= R_in(24);
        R_out(34)  <= R_in(23);
        R_out(33)  <= R_in(22);
        R_out(32)  <= R_in(21);
        R_out(31)  <= R_in(20);
        R_out(30)  <= R_in(19);
        R_out(29)  <= R_in(20);
        R_out(28)  <= R_in(19);
        R_out(27)  <= R_in(18);
        R_out(26)  <= R_in(17);
        R_out(25)  <= R_in(16);
        R_out(24)  <= R_in(15);
        R_out(23)  <= R_in(16);
        R_out(22)  <= R_in(15);
        R_out(21)  <= R_in(14);
        R_out(20)  <= R_in(13);
        R_out(19)  <= R_in(12);
        R_out(18)  <= R_in(11);
        R_out(17)  <= R_in(12);
        R_out(16)  <= R_in(11);
        R_out(15)  <= R_in(10);
        R_out(14)  <= R_in(9);
        R_out(13)  <= R_in(8);
        R_out(12)  <= R_in(7);
        R_out(11)  <= R_in(8);
        R_out(10)  <= R_in(7);
        R_out(9)   <= R_in(6);
```

```
        R_out(8)    <= R_in(5);
        R_out(7)    <= R_in(4);
        R_out(6)    <= R_in(3);
        R_out(5)    <= R_in(4);
        R_out(4)    <= R_in(3);
        R_out(3)    <= R_in(2);
        R_out(2)    <= R_in(1);
        R_out(1)    <= R_in(0);
        R_out(0)    <= R_in(31);
END comport;
--------------------------------------------------------------------------------


LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY XOR_32 IS
        PORT ( A : in  std_logic_vector ( 31 downto 0 );
               B : in  std_logic_vector ( 31 downto 0 );
               C : out std_logic_vector ( 31 downto 0 ));
END XOR_32;

ARCHITECTURE comport OF XOR_32 IS

BEGIN
        process (A, B)
        BEGIN
           C <= A XOR B;
        END process;
END comport;
 ------------------------------------------------------------------


LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY XOR_48 IS
        PORT ( A : in  std_logic_vector ( 47 downto 0 );
               B : in  std_logic_vector ( 47 downto 0 );
               C : out std_logic_vector ( 47 downto 0 ));
END XOR_48;

ARCHITECTURE comport OF XOR_48 IS

BEGIN
        process (A, B)
        BEGIN
           C <= A XOR B;
        END process;
END comport;
 --------------------------------------------------------------------------------
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY S_box IS
```



```
        PORT ( DIN  : in  std_logic_vector ( 47 downto 0 );
               DOUT : out std_logic_vector ( 31 downto 0 ));
END S_box;


ARCHITECTURE comport OF S_box IS

   SIGNAL S1, S2, S3, S4, S5, S6, S7, S8  : std_logic_vector ( 5 downto 0 );
   SIGNAL C1, C2, C3, C4, C5, C6, C7, C8  : std_logic_vector ( 3 downto 0 );

BEGIN
        S8(0) <= DIN(0);
        S8(1) <= DIN(1);
        S8(2) <= DIN(2);
        S8(3) <= DIN(3);
        S8(4) <= DIN(4);
        S8(5) <= DIN(5);


        S7(0) <= DIN(6);
        S7(1) <= DIN(7);
        S7(2) <= DIN(8);
        S7(3) <= DIN(9);
        S7(4) <= DIN(10);
        S7(5) <= DIN(11);

        S6(0) <= DIN(12);
        S6(1) <= DIN(13);
        S6(2) <= DIN(14);
        S6(3) <= DIN(15);
        S6(4) <= DIN(16);
        S6(5) <= DIN(17);

        S5(0) <= DIN(18);
        S5(1) <= DIN(19);
        S5(2) <= DIN(20);
        S5(3) <= DIN(21);
        S5(4) <= DIN(22);
        S5(5) <= DIN(23);

        S4(0) <= DIN(24);
        S4(1) <= DIN(25);
        S4(2) <= DIN(26);
        S4(3) <= DIN(27);
        S4(4) <= DIN(28);
        S4(5) <= DIN(29);

        S3(0) <= DIN(30);
        S3(1) <= DIN(31);
        S3(2) <= DIN(32);
        S3(3) <= DIN(33);
        S3(4) <= DIN(34);
        S3(5) <= DIN(35);

        S2(0) <= DIN(36);
        S2(1) <= DIN(37);
        S2(2) <= DIN(38);
        S2(3) <= DIN(39);
        S2(4) <= DIN(40);
        S2(5) <= DIN(41);
```

```
S1(0) <= DIN(42);
S1(1) <= DIN(43);
S1(2) <= DIN(44);
S1(3) <= DIN(45);
S1(4) <= DIN(46);
S1(5) <= DIN(47);


C1_process : PROCESS (S1)
BEGIN
        case S1 is
        when "000000" => C1 <= "1110";
        when "000010" => C1 <= "0100";
        when "000100" => C1 <= "1101";
        when "000110" => C1 <= "0001";
        when "001000" => C1 <= "0010";
        when "001010" => C1 <= "1111";
        when "001100" => C1 <= "1011";
        when "001110" => C1 <= "1000";
        when "010000" => C1 <= "0011";
        when "010010" => C1 <= "1010";
        when "010100" => C1 <= "0110";
        when "010110" => C1 <= "1100";
        when "011000" => C1 <= "0101";
        when "011010" => C1 <= "1001";
        when "011100" => C1 <= "0000";
        when "011110" => C1 <= "0111";

        when "000001" => C1 <= "0000";
        when "000011" => C1 <= "1111";
        when "000101" => C1 <= "0111";
        when "000111" => C1 <= "0100";
        when "001001" => C1 <= "1110";
        when "001011" => C1 <= "0010";
        when "001101" => C1 <= "1101";
        when "001111" => C1 <= "0001";
        when "010001" => C1 <= "1100";
        when "010011" => C1 <= "0110";
        when "010101" => C1 <= "1100";
        when "010111" => C1 <= "1011";
        when "011001" => C1 <= "1001";
        when "011011" => C1 <= "0101";
        when "011101" => C1 <= "0011";
        when "011111" => C1 <= "1000";

        when "100000" => C1 <= "0100";
        when "100010" => C1 <= "0001";
        when "100100" => C1 <= "1110";
        when "100110" => C1 <= "1000";
        when "101000" => C1 <= "1101";
        when "101010" => C1 <= "0110";
        when "101100" => C1 <= "0010";
        when "101110" => C1 <= "1011";
        when "110000" => C1 <= "1111";
        when "110010" => C1 <= "1100";
        when "110100" => C1 <= "1001";
        when "110110" => C1 <= "0111";
        when "111000" => C1 <= "0011";
        when "111010" => C1 <= "1010";
```



```
        when "111100" => C1 <= "0101";
        when "111110" => C1 <= "0000";

        when "100001" => C1 <= "1111";
        when "100011" => C1 <= "1100";
        when "100101" => C1 <= "1000";
        when "100111" => C1 <= "0010";
        when "101001" => C1 <= "0100";
        when "101011" => C1 <= "1001";
        when "101101" => C1 <= "0001";
        when "101111" => C1 <= "0111";
        when "110001" => C1 <= "0101";
        when "110011" => C1 <= "1011";
        when "110101" => C1 <= "0011";
        when "110111" => C1 <= "1110";
        when "111001" => C1 <= "1010";
        when "111011" => C1 <= "0000";
        when "111101" => C1 <= "0110";
        when "111111" => C1 <= "1101";

        when others => C1 <= "1111";

        END case;
END PROCESS C1_process;


C2_process : PROCESS (S2)
BEGIN

        case S2 is
        when "000000" => C2 <= "1111";
        when "000010" => C2 <= "0001";
        when "000100" => C2 <= "1000";
        when "000110" => C2 <= "1110";
        when "001000" => C2 <= "0110";
        when "001010" => C2 <= "1011";
        when "001100" => C2 <= "0011";
        when "001110" => C2 <= "0100";
        when "010000" => C2 <= "1001";
        when "010010" => C2 <= "0111";
        when "010100" => C2 <= "0010";
        when "010110" => C2 <= "1101";
        when "011000" => C2 <= "1100";
        when "011010" => C2 <= "0000";
        when "011100" => C2 <= "0101";
        when "011110" => C2 <= "1010";

        when "000001" => C2 <= "0011";
        when "000011" => C2 <= "1101";
        when "000101" => C2 <= "0100";
        when "000111" => C2 <= "0111";
        when "001001" => C2 <= "1111";
        when "001011" => C2 <= "0010";
        when "001101" => C2 <= "1000";
        when "001111" => C2 <= "1110";
        when "010001" => C2 <= "1100";
        when "010011" => C2 <= "0000";
        when "010101" => C2 <= "0001";
        when "010111" => C2 <= "1010";
        when "011001" => C2 <= "0110";
```

```
        when "011011" => C2 <= "1001";
        when "011101" => C2 <= "1011";
        when "011111" => C2 <= "0101";


        when "100000" => C2 <= "0000";
        when "100010" => C2 <= "1110";
        when "100100" => C2 <= "0111";
        when "100110" => C2 <= "1011";
        when "101000" => C2 <= "1010";
        when "101010" => C2 <= "0100";
        when "101100" => C2 <= "1101";
        when "101110" => C2 <= "0001";
        when "110000" => C2 <= "0101";
        when "110010" => C2 <= "1000";
        when "110100" => C2 <= "1100";
        when "110110" => C2 <= "0110";
        when "111000" => C2 <= "1001";
        when "111010" => C2 <= "0011";
        when "111100" => C2 <= "0010";
        when "111110" => C2 <= "1111";

        when "100001" => C2 <= "1101";
        when "100011" => C2 <= "1000";
        when "100101" => C2 <= "1010";
        when "100111" => C2 <= "0001";
        when "101001" => C2 <= "0011";
        when "101011" => C2 <= "1111";
        when "101101" => C2 <= "0100";
        when "101111" => C2 <= "0010";
        when "110001" => C2 <= "1011";
        when "110011" => C2 <= "0110";
        when "110101" => C2 <= "0111";
        when "110111" => C2 <= "1100";
        when "111001" => C2 <= "0000";
        when "111011" => C2 <= "0101";
        when "111101" => C2 <= "1110";
        when "111111" => C2 <= "1001";

        when others => C2 <= "1111";
        END case;
END PROCESS C2_process;

C3_process : PROCESS (S3)
BEGIN

        case S3 is
        when "000000" => C3 <= "1010";
        when "000010" => C3 <= "0000";
        when "000100" => C3 <= "1001";
        when "000110" => C3 <= "1110";
        when "001000" => C3 <= "0110";
        when "001010" => C3 <= "0011";
        when "001100" => C3 <= "1111";
        when "001110" => C3 <= "0101";
        when "010000" => C3 <= "0001";
        when "010010" => C3 <= "1101";
        when "010100" => C3 <= "1100";
        when "010110" => C3 <= "0111";
        when "011000" => C3 <= "1011";
```



```
        when "011010" => C3 <= "0100";
        when "011100" => C3 <= "0010";
        when "011110" => C3 <= "1000";

        when "000001" => C3 <= "1101";
        when "000011" => C3 <= "0111";
        when "000101" => C3 <= "0000";
        when "000111" => C3 <= "1001";
        when "001001" => C3 <= "0011";
        when "001011" => C3 <= "0100";
        when "001101" => C3 <= "0110";
        when "001111" => C3 <= "1010";
        when "010001" => C3 <= "0010";
        when "010011" => C3 <= "1000";
        when "010101" => C3 <= "0101";
        when "010111" => C3 <= "1110";
        when "011001" => C3 <= "1100";
        when "011011" => C3 <= "1011";
        when "011101" => C3 <= "1111";
        when "011111" => C3 <= "0001";


        when "100000" => C3 <= "1101";
        when "100010" => C3 <= "0110";
        when "100100" => C3 <= "0100";
        when "100110" => C3 <= "1001";
        when "101000" => C3 <= "1000";
        when "101010" => C3 <= "1111";
        when "101100" => C3 <= "0011";
        when "101110" => C3 <= "0000";
        when "110000" => C3 <= "1011";
        when "110010" => C3 <= "0001";
        when "110100" => C3 <= "0011";
        when "110110" => C3 <= "1100";
        when "111000" => C3 <= "0101";
        when "111010" => C3 <= "1010";
        when "111100" => C3 <= "1110";
        when "111110" => C3 <= "0111";

        when "100001" => C3 <= "0001";
        when "100011" => C3 <= "1010";
        when "100101" => C3 <= "1101";
        when "100111" => C3 <= "0000";
        when "101001" => C3 <= "0110";
        when "101011" => C3 <= "1001";
        when "101101" => C3 <= "1000";
        when "101111" => C3 <= "0111";
        when "110001" => C3 <= "1100";
        when "110011" => C3 <= "1111";
        when "110101" => C3 <= "1110";
        when "110111" => C3 <= "0011";
        when "111001" => C3 <= "1011";
        when "111011" => C3 <= "0101";
        when "111101" => C3 <= "0010";
        when "111111" => C3 <= "1100";

        when others => C3 <= "1111";
        END case;

END PROCESS C3_process;
```

```
C4_process : PROCESS (S4)
BEGIN
        case S4 is
        when "000000" => C4 <= "0111";
        when "000010" => C4 <= "1101";
        when "000100" => C4 <= "1110";
        when "000110" => C4 <= "0011";
        when "001000" => C4 <= "0000";
        when "001010" => C4 <= "0110";
        when "001100" => C4 <= "1001";
        when "001110" => C4 <= "1010";
        when "010000" => C4 <= "0001";
        when "010010" => C4 <= "0010";
        when "010100" => C4 <= "1000";
        when "010110" => C4 <= "0101";
        when "011000" => C4 <= "1011";
        when "011010" => C4 <= "1100";
        when "011100" => C4 <= "0100";
        when "011110" => C4 <= "1111";

        when "000001" => C4 <= "1101";
        when "000011" => C4 <= "1000";
        when "000101" => C4 <= "1011";
        when "000111" => C4 <= "0101";
        when "001001" => C4 <= "0110";
        when "001011" => C4 <= "1111";
        when "001101" => C4 <= "0000";
        when "001111" => C4 <= "0011";
        when "010001" => C4 <= "0100";
        when "010011" => C4 <= "0111";
        when "010101" => C4 <= "0010";
        when "010111" => C4 <= "1100";
        when "011001" => C4 <= "0001";
        when "011011" => C4 <= "1010";
        when "011101" => C4 <= "1110";
        when "011111" => C4 <= "1001";


        when "100000" => C4 <= "1010";
        when "100010" => C4 <= "0110";
        when "100100" => C4 <= "1001";
        when "100110" => C4 <= "0000";
        when "101000" => C4 <= "1100";
        when "101010" => C4 <= "1011";
        when "101100" => C4 <= "0111";
        when "101110" => C4 <= "1101";
        when "110000" => C4 <= "1111";
        when "110010" => C4 <= "0001";
        when "110100" => C4 <= "0011";
        when "110110" => C4 <= "1110";
        when "111000" => C4 <= "0101";
        when "111010" => C4 <= "0010";
        when "111100" => C4 <= "1000";
        when "111110" => C4 <= "0100";

        when "100001" => C4 <= "0011";
        when "100011" => C4 <= "1111";
        when "100101" => C4 <= "0000";
```



```
        when "100111" => C4 <= "0110";
        when "101001" => C4 <= "1010";
        when "101011" => C4 <= "0001";
        when "101101" => C4 <= "1101";
        when "101111" => C4 <= "1000";
        when "110001" => C4 <= "1001";
        when "110011" => C4 <= "0100";
        when "110101" => C4 <= "0101";
        when "110111" => C4 <= "1011";
        when "111001" => C4 <= "1100";
        when "111011" => C4 <= "0111";
        when "111101" => C4 <= "0010";
        when "111111" => C4 <= "1110";

        when others => C4 <= "1111";
        END case;

END PROCESS C4_process;


C5_process : PROCESS (S5)
BEGIN
        case S5 is
        when "000000" => C5 <= "0010";
        when "000010" => C5 <= "1100";
        when "000100" => C5 <= "0100";
        when "000110" => C5 <= "0001";
        when "001000" => C5 <= "0111";
        when "001010" => C5 <= "1010";
        when "001100" => C5 <= "1011";
        when "001110" => C5 <= "0110";
        when "010000" => C5 <= "1000";
        when "010010" => C5 <= "0101";
        when "010100" => C5 <= "0011";
        when "010110" => C5 <= "1111";
        when "011000" => C5 <= "1101";
        when "011010" => C5 <= "0000";
        when "011100" => C5 <= "1110";
        when "011110" => C5 <= "1001";

        when "000001" => C5 <= "1110";
        when "000011" => C5 <= "1011";
        when "000101" => C5 <= "0010";
        when "000111" => C5 <= "1100";
        when "001001" => C5 <= "0100";
        when "001011" => C5 <= "0111";
        when "001101" => C5 <= "1101";
        when "001111" => C5 <= "0001";
        when "010001" => C5 <= "0101";
        when "010011" => C5 <= "0000";
        when "010101" => C5 <= "1111";
        when "010111" => C5 <= "1010";
        when "011001" => C5 <= "0011";
        when "011011" => C5 <= "1001";
        when "011101" => C5 <= "1000";
        when "011111" => C5 <= "0110";


        when "100000" => C5 <= "0100";
```

```
        when "100010" => C5 <= "0010";
        when "100100" => C5 <= "0001";
        when "100110" => C5 <= "1011";
        when "101000" => C5 <= "1010";
        when "101010" => C5 <= "1101";
        when "101100" => C5 <= "0111";
        when "101110" => C5 <= "1000";
        when "110000" => C5 <= "1111";
        when "110010" => C5 <= "1001";
        when "110100" => C5 <= "1100";
        when "110110" => C5 <= "0101";
        when "111000" => C5 <= "0110";
        when "111010" => C5 <= "0011";
        when "111100" => C5 <= "0000";
        when "111110" => C5 <= "1110";

        when "100001" => C5 <= "1011";
        when "100011" => C5 <= "1000";
        when "100101" => C5 <= "1100";
        when "100111" => C5 <= "0111";
        when "101001" => C5 <= "0001";
        when "101011" => C5 <= "1110";
        when "101101" => C5 <= "0010";
        when "101111" => C5 <= "1101";
        when "110001" => C5 <= "0110";
        when "110011" => C5 <= "1111";
        when "110101" => C5 <= "0000";
        when "110111" => C5 <= "1001";
        when "111001" => C5 <= "1010";
        when "111011" => C5 <= "0100";
        when "111101" => C5 <= "0101";
        when "111111" => C5 <= "0011";

        when others => C5 <= "1111";
        END case;

END PROCESS C5_process;


C6_process : PROCESS (S6)
BEGIN
        case S6 is
        when "000000" => C6 <= "1100";
        when "000010" => C6 <= "0001";
        when "000100" => C6 <= "1010";
        when "000110" => C6 <= "1111";
        when "001000" => C6 <= "1001";
        when "001010" => C6 <= "0010";
        when "001100" => C6 <= "0110";
        when "001110" => C6 <= "1000";
        when "010000" => C6 <= "0000";
        when "010010" => C6 <= "1101";
        when "010100" => C6 <= "0011";
        when "010110" => C6 <= "0100";
        when "011000" => C6 <= "1110";
        when "011010" => C6 <= "0111";
        when "011100" => C6 <= "0101";
        when "011110" => C6 <= "1011";

        when "000001" => C6 <= "1010";
```



```
        when "000011" => C6 <= "1111";
        when "000101" => C6 <= "0100";
        when "000111" => C6 <= "0010";
        when "001001" => C6 <= "0111";
        when "001011" => C6 <= "1100";
        when "001101" => C6 <= "1001";
        when "001111" => C6 <= "0101";
        when "010001" => C6 <= "0110";
        when "010011" => C6 <= "0001";
        when "010101" => C6 <= "1101";
        when "010111" => C6 <= "1110";
        when "011001" => C6 <= "0000";
        when "011011" => C6 <= "1011";
        when "011101" => C6 <= "0011";
        when "011111" => C6 <= "1000";


        when "100000" => C6 <= "1001";
        when "100010" => C6 <= "1110";
        when "100100" => C6 <= "1111";
        when "100110" => C6 <= "0101";
        when "101000" => C6 <= "0010";
        when "101010" => C6 <= "1000";
        when "101100" => C6 <= "1100";
        when "101110" => C6 <= "0011";
        when "110000" => C6 <= "0111";
        when "110010" => C6 <= "0000";
        when "110100" => C6 <= "0100";
        when "110110" => C6 <= "1010";
        when "111000" => C6 <= "0001";
        when "111010" => C6 <= "1101";
        when "111100" => C6 <= "1011";
        when "111110" => C6 <= "0110";

        when "100001" => C6 <= "0100";
        when "100011" => C6 <= "0011";
        when "100101" => C6 <= "0010";
        when "100111" => C6 <= "1100";
        when "101001" => C6 <= "1001";
        when "101011" => C6 <= "1101";
        when "101101" => C6 <= "1111";
        when "101111" => C6 <= "1010";
        when "110001" => C6 <= "1011";
        when "110011" => C6 <= "1110";
        when "110101" => C6 <= "0001";
        when "110111" => C6 <= "0111";
        when "111001" => C6 <= "0110";
        when "111011" => C6 <= "0000";
        when "111101" => C6 <= "1000";
        when "111111" => C6 <= "1101";

        when others => C6 <= "1111";
        END case;

END PROCESS C6_process;


C7_process : PROCESS (S7)
BEGIN
        case S7 is
```

```
when "000000" => C7 <= "0100";
when "000010" => C7 <= "1011";
when "000100" => C7 <= "0010";
when "000110" => C7 <= "1110";
when "001000" => C7 <= "1111";
when "001010" => C7 <= "0000";
when "001100" => C7 <= "1000";
when "001110" => C7 <= "1101";
when "010000" => C7 <= "0011";
when "010010" => C7 <= "1100";
when "010100" => C7 <= "1001";
when "010110" => C7 <= "0111";
when "011000" => C7 <= "0101";
when "011010" => C7 <= "1010";
when "011100" => C7 <= "0110";
when "011110" => C7 <= "0001";

when "000001" => C7 <= "1101";
when "000011" => C7 <= "0000";
when "000101" => C7 <= "1011";
when "000111" => C7 <= "0111";
when "001001" => C7 <= "0100";
when "001011" => C7 <= "1001";
when "001101" => C7 <= "0001";
when "001111" => C7 <= "1010";
when "010001" => C7 <= "1110";
when "010011" => C7 <= "0011";
when "010101" => C7 <= "0101";
when "010111" => C7 <= "1100";
when "011001" => C7 <= "0010";
when "011011" => C7 <= "1111";
when "011101" => C7 <= "1000";
when "011111" => C7 <= "0110";


when "100000" => C7 <= "0001";
when "100010" => C7 <= "0100";
when "100100" => C7 <= "1011";
when "100110" => C7 <= "1101";
when "101000" => C7 <= "1100";
when "101010" => C7 <= "0011";
when "101100" => C7 <= "0111";
when "101110" => C7 <= "1110";
when "110000" => C7 <= "1010";
when "110010" => C7 <= "1111";
when "110100" => C7 <= "0110";
when "110110" => C7 <= "1000";
when "111000" => C7 <= "0000";
when "111010" => C7 <= "0101";
when "111100" => C7 <= "1001";
when "111110" => C7 <= "0010";

when "100001" => C7 <= "0110";
when "100011" => C7 <= "1011";
when "100101" => C7 <= "1101";
when "100111" => C7 <= "1000";
when "101001" => C7 <= "0001";
when "101011" => C7 <= "0100";
when "101101" => C7 <= "1010";
when "101111" => C7 <= "0111";
```



```
when "110001" => C7 <= "1001";
when "110011" => C7 <= "0101";
when "110101" => C7 <= "0000";
when "110111" => C7 <= "1111";
when "111001" => C7 <= "1110";
when "111011" => C7 <= "0010";
when "111101" => C7 <= "0011";
when "111111" => C7 <= "1100";

when others => C7 <= "1111";
END case;

END PROCESS C7_process;


C8_process : PROCESS (S8)
BEGIN
        case S8 is
        when "000000" => C8 <= "1101";
        when "000010" => C8 <= "0010";
        when "000100" => C8 <= "1000";
        when "000110" => C8 <= "0100";
        when "001000" => C8 <= "0110";
        when "001010" => C8 <= "1111";
        when "001100" => C8 <= "1011";
        when "001110" => C8 <= "0001";
        when "010000" => C8 <= "1010";
        when "010010" => C8 <= "1001";
        when "010100" => C8 <= "0011";
        when "010110" => C8 <= "1110";
        when "011000" => C8 <= "0101";
        when "011010" => C8 <= "0000";
        when "011100" => C8 <= "1100";
        when "011110" => C8 <= "0111";

        when "000001" => C8 <= "0001";
        when "000011" => C8 <= "1111";
        when "000101" => C8 <= "1101";
        when "000111" => C8 <= "1000";
        when "001001" => C8 <= "1010";
        when "001011" => C8 <= "0011";
        when "001101" => C8 <= "0111";
        when "001111" => C8 <= "0100";
        when "010001" => C8 <= "1100";
        when "010011" => C8 <= "0101";
        when "010101" => C8 <= "0110";
        when "010111" => C8 <= "1011";
        when "011001" => C8 <= "0000";
        when "011011" => C8 <= "1110";
        when "011101" => C8 <= "1001";
        when "011111" => C8 <= "0010";


        when "100000" => C8 <= "0111";
        when "100010" => C8 <= "1011";
        when "100100" => C8 <= "0100";
        when "100110" => C8 <= "0001";
        when "101000" => C8 <= "1001";
        when "101010" => C8 <= "1100";
        when "101100" => C8 <= "1110";
```

```
            when "101110" => C8 <= "0010";
            when "110000" => C8 <= "0000";
            when "110010" => C8 <= "0110";
            when "110100" => C8 <= "1010";
            when "110110" => C8 <= "1101";
            when "111000" => C8 <= "1111";
            when "111010" => C8 <= "0011";
            when "111100" => C8 <= "0101";
            when "111110" => C8 <= "1000";

            when "100001" => C8 <= "0010";
            when "100011" => C8 <= "0001";
            when "100101" => C8 <= "1110";
            when "100111" => C8 <= "0111";
            when "101001" => C8 <= "0100";
            when "101011" => C8 <= "1010";
            when "101101" => C8 <= "1000";
            when "101111" => C8 <= "1101";
            when "110001" => C8 <= "1111";
            when "110011" => C8 <= "1100";
            when "110101" => C8 <= "1001";
            when "110111" => C8 <= "0000";
            when "111001" => C8 <= "0011";
            when "111011" => C8 <= "0101";
            when "111101" => C8 <= "0110";
            when "111111" => C8 <= "1101";

            when others => C8 <= "1111";
            END case;

    END PROCESS C8_process;


    DOUT(31) <= C4(0);
    DOUT(30) <= C2(1);
    DOUT(29) <= C5(0);
    DOUT(28) <= C6(3);
    DOUT(27) <= C8(3);
    DOUT(26) <= C3(0);
    DOUT(25) <= C7(0);
    DOUT(24) <= C5(3);
    DOUT(23) <= C1(3);
    DOUT(22) <= C4(1);
    DOUT(21) <= C6(1);
    DOUT(20) <= C7(2);
    DOUT(19) <= C2(3);
    DOUT(18) <= C5(2);
    DOUT(17) <= C8(1);
    DOUT(16) <= C3(2);
    DOUT(15) <= C1(2);
    DOUT(14) <= C2(0);
    DOUT(13) <= C6(0);
    DOUT(12) <= C4(2);
    DOUT(11) <= C8(0);
    DOUT(10) <= C7(1);
    DOUT(9)  <= C1(1);
    DOUT(8)  <= C3(3);
    DOUT(7)  <= C5(1);
    DOUT(6)  <= C4(3);
    DOUT(5)  <= C8(2);
```



```
    DOUT(4)  <= C2(2);
    DOUT(3)  <= C6(2);
    DOUT(2)  <= C3(1);
    DOUT(1)  <= C1(0);
    DOUT(0)  <= C7(3);

END comport;
-------------------------------------------------------

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY S_Generater IS
        PORT( CLK          : in  std_logic;
              CLK_16       : out std_logic;
              start_enc    : out std_logic;
              start_out    : out std_logic);
END S_Generater;

architecture behavour of S_Generater is

begin
   U_1 :process
   variable count : integer range 1 to 127;
   begin
        wait until (CLK' event and CLK = '1');
        if(count < 64) then
             start_enc <= '0';
             count := count + 1;
        elsif(count = 64) then
             start_enc <= '1';
             count := count + 1;
        elsif((count >= 65) and (count < 96 )) then
             start_enc <= '1';
             count := count + 1;
        elsif((count >= 96) and (count < 127 )) then
             start_enc <= '0';
             count := count + 1;
         elsif(count = 127) then
             start_enc <= '0';
             count := 64;
        end if;
   end process;

   U_2 :process
   variable count1 : integer range 0 to 127;
   begin
        wait until (CLK' event and CLK = '1');
        if(count1 < 127) then
             count1 := count1 + 1;
             start_out <= '0';
        elsif(count1 = 127) then
             start_out <= '1';
             count1 := 64;
        end if;
   end process;

   U_3 :process
   variable count1 : integer range 1 to 2;
```

```
    begin
        wait until (CLK' event and CLK = '1');
        if(count1 < 2) then
            CLK_16 <= '1';
            count1 := count1 + 1;

        elsif(count1 = 2) then
            CLK_16 <= '0';
            count1 := 1;
        end if;
    end process;
end behaveur;

-------------------------------------------------------------

LIBRARY ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY OUT_D IS
        PORT( out_din       : in std_logic_vector ( 63 downto 0 );
              CLK           : in std_logic;
              start_out     : in std_logic;
              out_dout      : out std_logic);
END OUT_D ;

ARCHITECTURE comport OF OUT_D IS
TYPE memor is array (63 downto 0 ) of std_logic;
begin
    U_1 : process
          VARIABLE data : memor;
        begin
           wait until ( CLK'event and CLK ='1');
           if (start_out = '1' ) then
               data(0) := out_din(39);
               data(1) := out_din(7);
               data(2) := out_din(47);
               data(3) := out_din(15);
               data(4) := out_din(55);
               data(5) := out_din(23);
               data(6) := out_din(63);
               data(7) := out_din(31);
               data(8) := out_din(38);
               data(9) := out_din(6);
               data(10) := out_din(46);
               data(11) := out_din(14);
               data(12) := out_din(54);
               data(13) := out_din(22);
               data(14) := out_din(62);
               data(15) := out_din(30);
               data(16) := out_din(37);
               data(17) := out_din(5);
               data(18) := out_din(45);
               data(19) := out_din(13);
               data(20) := out_din(53);
               data(21) := out_din(21);
               data(22) := out_din(61);
               data(23) := out_din(29);
               data(24) := out_din(36);
               data(25) := out_din(4);
```



```
               data(26) := out_din(44);
               data(27) := out_din(12);
               data(28) := out_din(52);
               data(29) := out_din(20);
               data(30) := out_din(60);
               data(31) := out_din(28);
               data(32) := out_din(35);
               data(33) := out_din(3);
               data(34) := out_din(43);
               data(35) := out_din(11);
               data(36) := out_din(51);
               data(37) := out_din(19);
               data(38) := out_din(59);
               data(39) := out_din(27);
               data(40) := out_din(34);
               data(41) := out_din(2);
               data(42) := out_din(42);
               data(43) := out_din(10);
               data(44) := out_din(50);
               data(45) := out_din(18);
               data(46) := out_din(58);
               data(47) := out_din(26);
               data(48) := out_din(33);
               data(49) := out_din(1);
               data(50) := out_din(41);
               data(51) := out_din(9);
               data(52) := out_din(49);
               data(53) := out_din(17);
               data(54) := out_din(57);
               data(55) := out_din(25);
               data(56) := out_din(32);
               data(57) := out_din(0);
               data(58) := out_din(40);
               data(59) := out_din(8);
               data(60) := out_din(48);
               data(61) := out_din(16);
               data(62) := out_din(56);
               data(63) := out_din(24);

           end if;
           out_dout <= data(63);

           for I in 62 downto 0 loop
                data (I+1) := data(I);
           end loop;

    end process;
end comport;
```

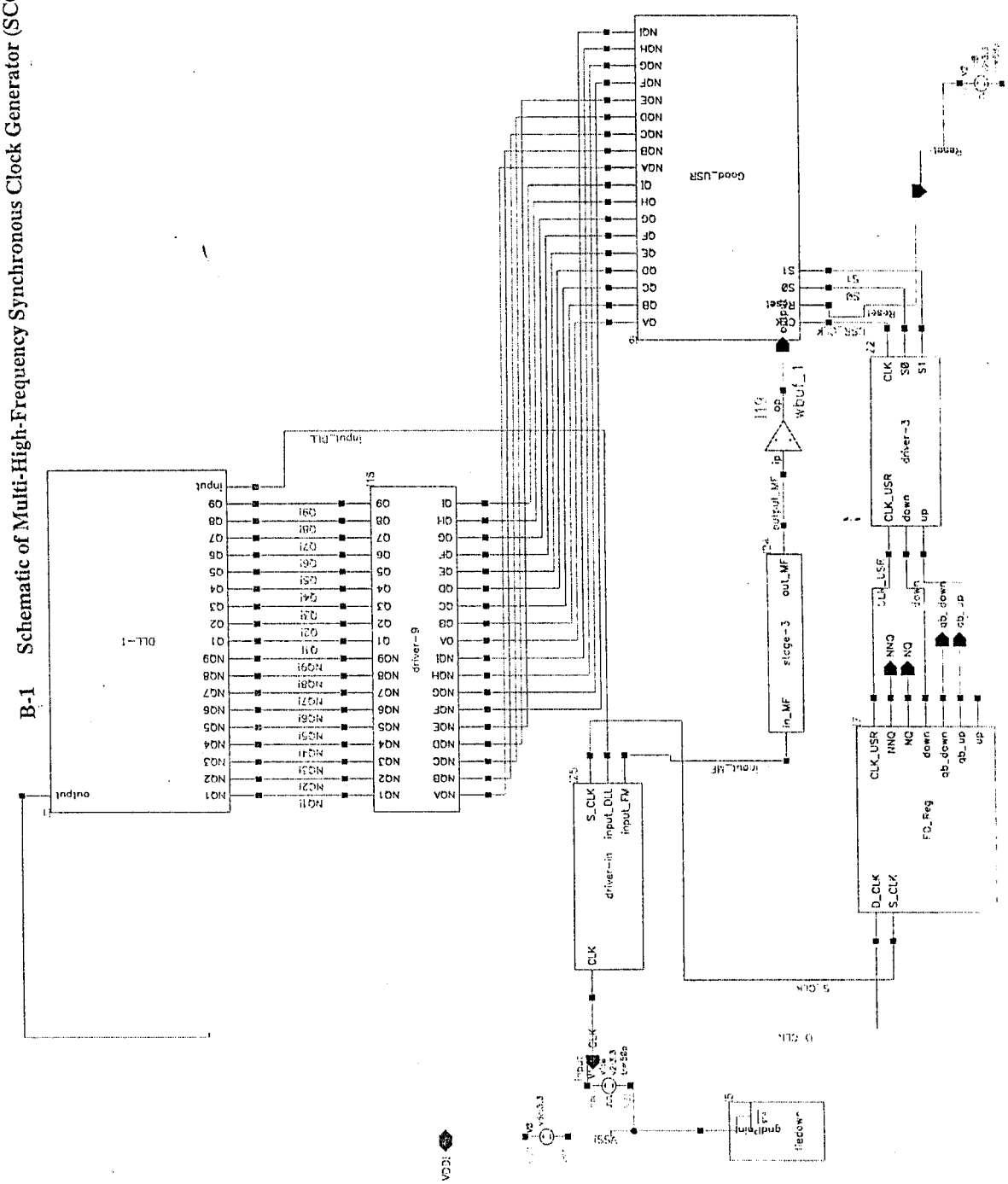
A-5 Script file for DES algorithm synthesis design on FPGA-based

```
/*-------------------------------------------------------------*/
/*     Sample Script for Synopsys to Xilinx Using              */
/*                  FPGA Compiler                              */
/*                                                             */
/*    Target the Xilinx 4028ex-2 and assumes a VHDL            */
/*    source file by way of an eaample.                        */
/*                                                             */
/*    For general use with XC4000E architectures.              */
/*                                                             */
/*-------------------------------------------------------------*/


TOP = TOP


designer = "Beisong Liu"
company = "Ecole Polytechnique"
part = "xc4028exhq304-2"

/*-------------------------------------------------------------*/
/*--  Analyze and Elaborate the design file and specify       */
/*--  the design file format.                                 */
/*-------------------------------------------------------------*/


analyze -format vhdl ./all.vhd
elaborate TOP
uniquify
set_max_area 0

/*-------------------------------------------------------------*/
/*-- Set the current design to the top level.                 */
/*-------------------------------------------------------------*/

current_design TOP

/*-------------------------------------------------------------*/
/*--  Set the synthesis design constraints.                   */
/*-------------------------------------------------------------*/

remove_constraint -all


/*-------------------------------------------------------------*/
/*--  Apply constraints to the synthesis process              */
/*-------------------------------------------------------------*/


create_clock Clock -period 50

/*-------------------------------------------------------------*/
/*-- Indicate those ports on the top-level module that        */
/*-- should become chip-level I/O pads. Assign any I/O        */
/*-- attributes or parameters and perform the I/O synthesis.  */
/*-------------------------------------------------------------*/
```

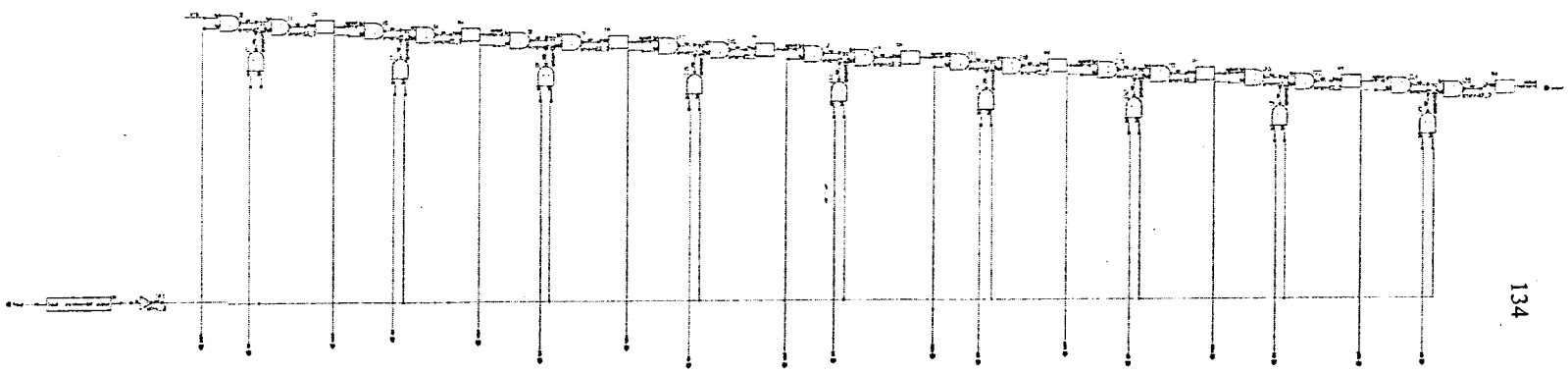
```
set_port_is_pad "*"

set_pad_type -no_clock all_inputs( )
set_pad_type -clock Clock
set_pad_type -slewrate HIGH all_Outputs( )
insert_pads -verify -verify_effort low

/*-----------------------------------------------------------------*/
/*-- Synthesize and optinmize the design                          */
/*-----------------------------------------------------------------*/

uniquify
compile -boundary_optimization


/*-----------------------------------------------------------------*/
/*--  Write out the design to a DB file. (before replace_fpga)     */
/*-----------------------------------------------------------------*/


write -format db -hierarchy -output TOP + "_before_fpga.db"

/*-----------------------------------------------------------------*/
/*--  Write the design report files                                */
/*-----------------------------------------------------------------*/

report_fpga > TOP + ".fpga"
report_timing > TOP + ".timing"

/*-----------------------------------------------------------------*/
/*--  Replace CLBs and IOBs with gates                             */
/*-----------------------------------------------------------------*/

uniquify
replace_fpga

/*-----------------------------------------------------------------*/
/*--  Setthe part type for the output netlist                      */
/*-----------------------------------------------------------------*/

set_attribute TOP "part" -type string part


/*-----------------------------------------------------------------*/
/*-- Save design in XNF format as <design>. sxnf                  */
/*-----------------------------------------------------------------*/

ungroup -all -flatten
write -format xnf -hierarchy -output TOP + ".sxnf"

/*-----------------------------------------------------------------*/
/*-- Write the design to a DB (Post replace_fpga)                 */
/*-----------------------------------------------------------------*/

write -format db -hierarchy -output TOP + "_post_replace_fpga.db"
```

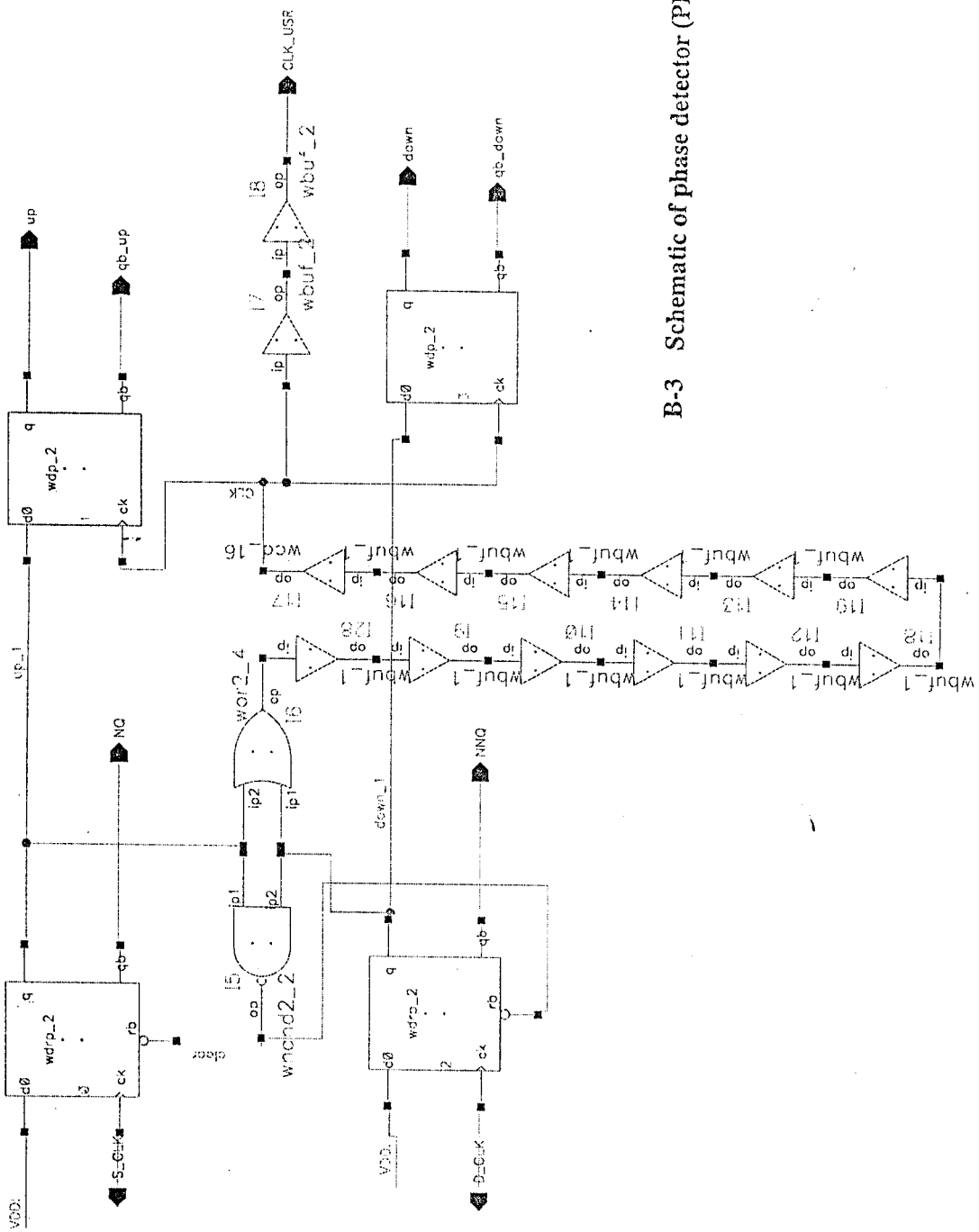
```
/*-------------------------------------------------------------------*/
/*--  Exit the compiler                                            */
/*-------------------------------------------------------------------*/

exit

/*-------------------------------------------------------------------*/
/*--  Now run the Xilinx design implementation tools               */
/*-------------------------------------------------------------------*/
```

B-1 Schematic of Multi-High-Frequency Synchronous Clock Generator (SCG)



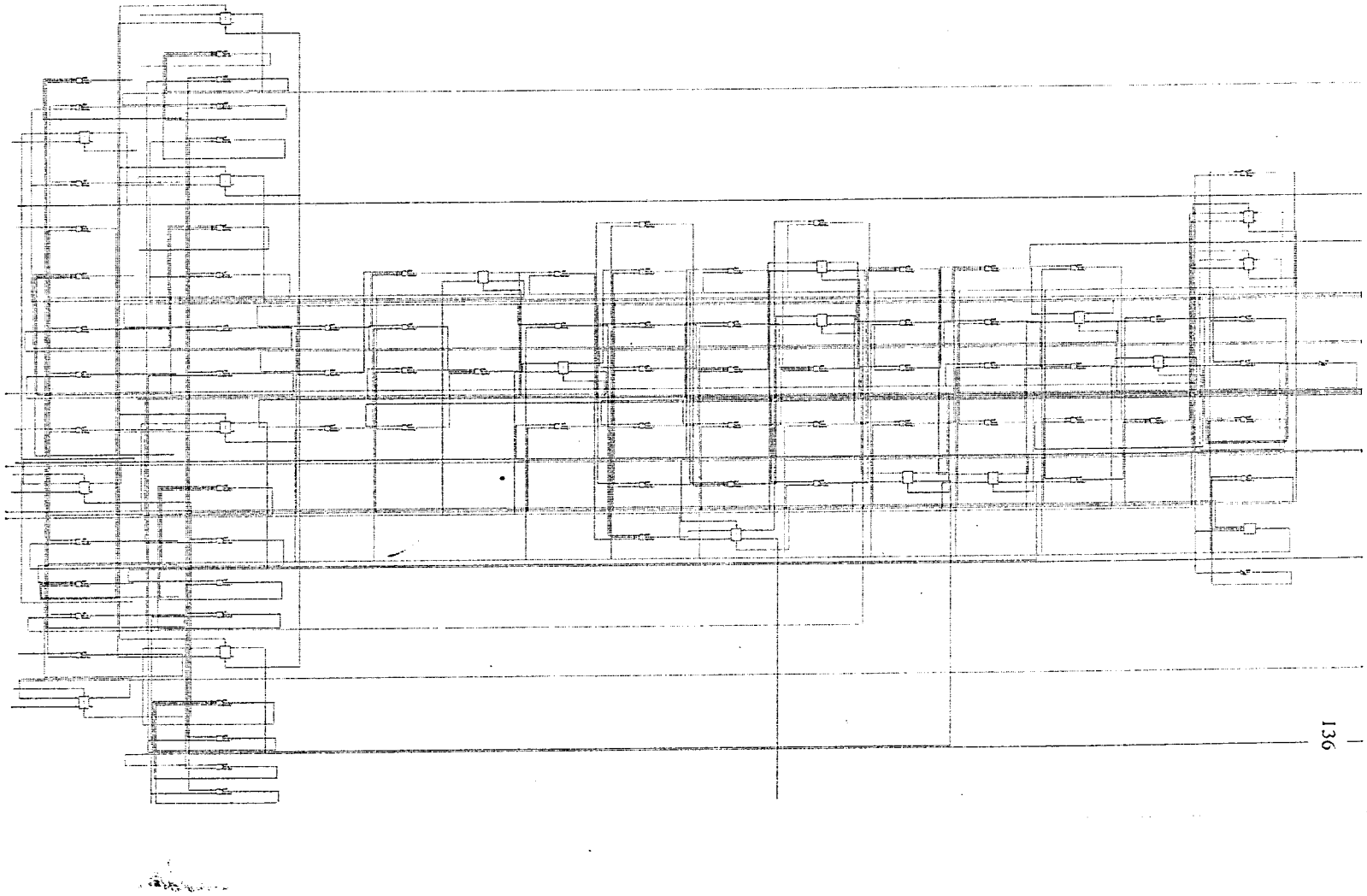
B-2 Schematic of delay line (DL)

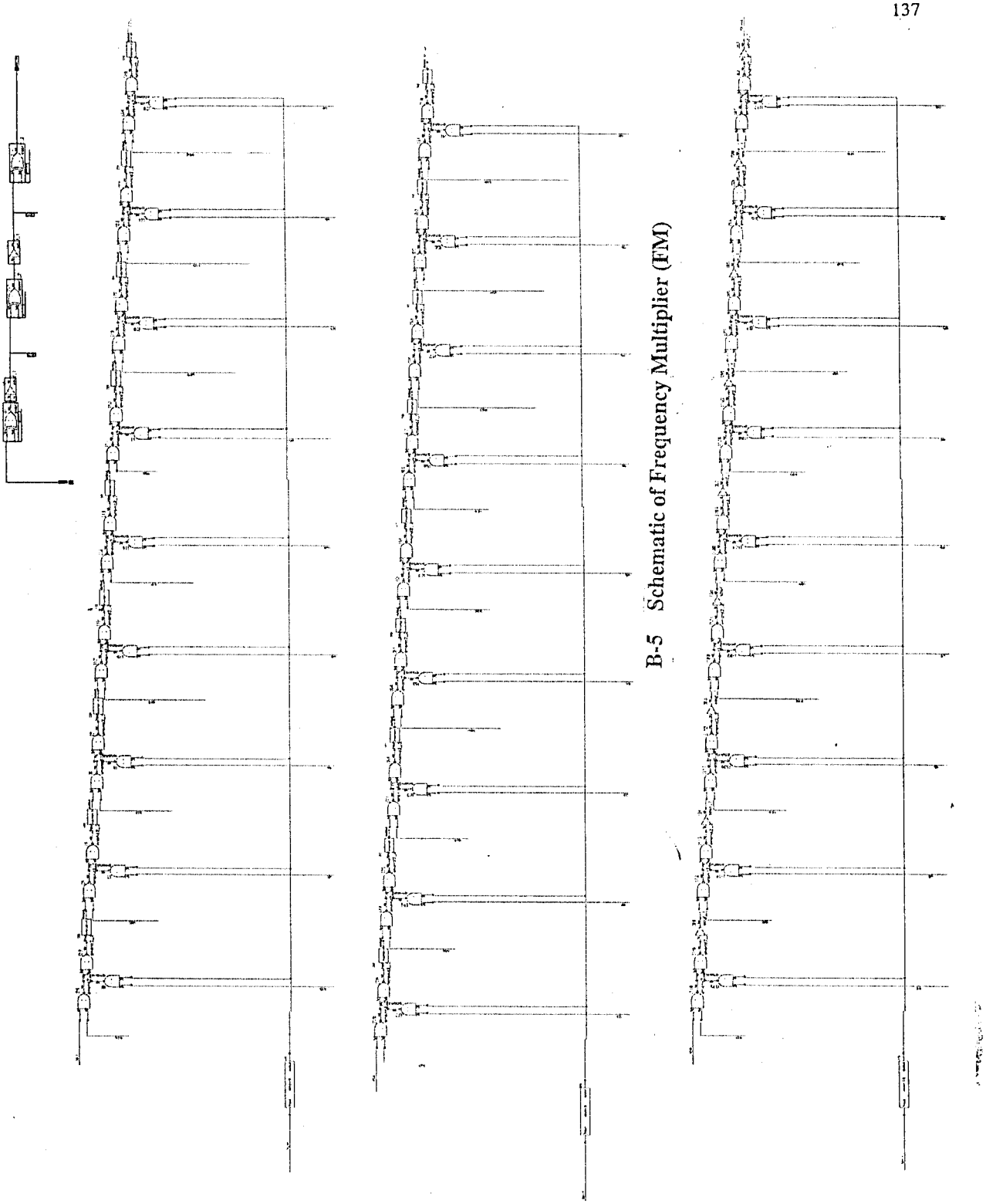




B-3 Schematic of phase detector (PD)

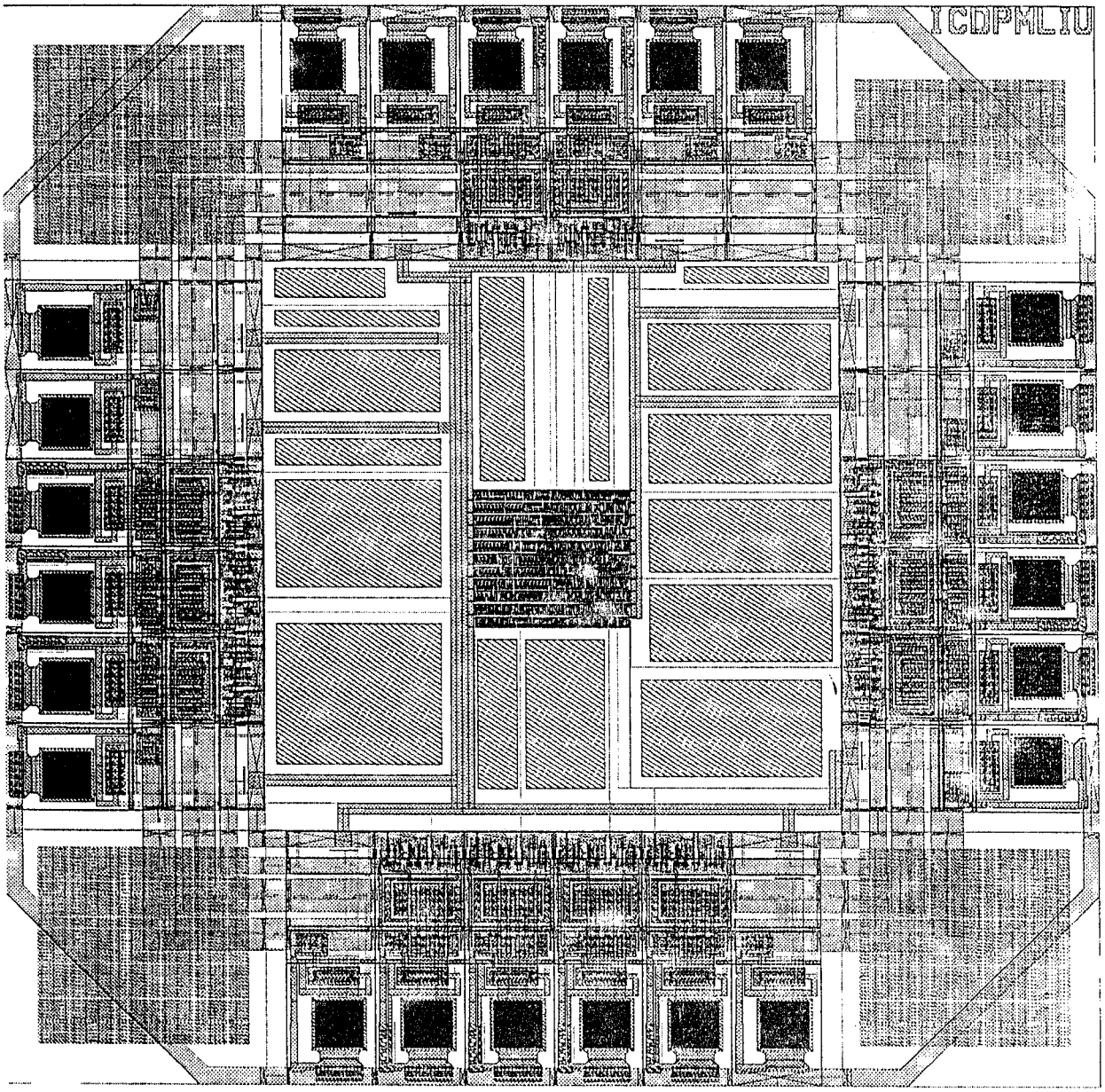
B-4 Schematic of universal shirt register (USR)



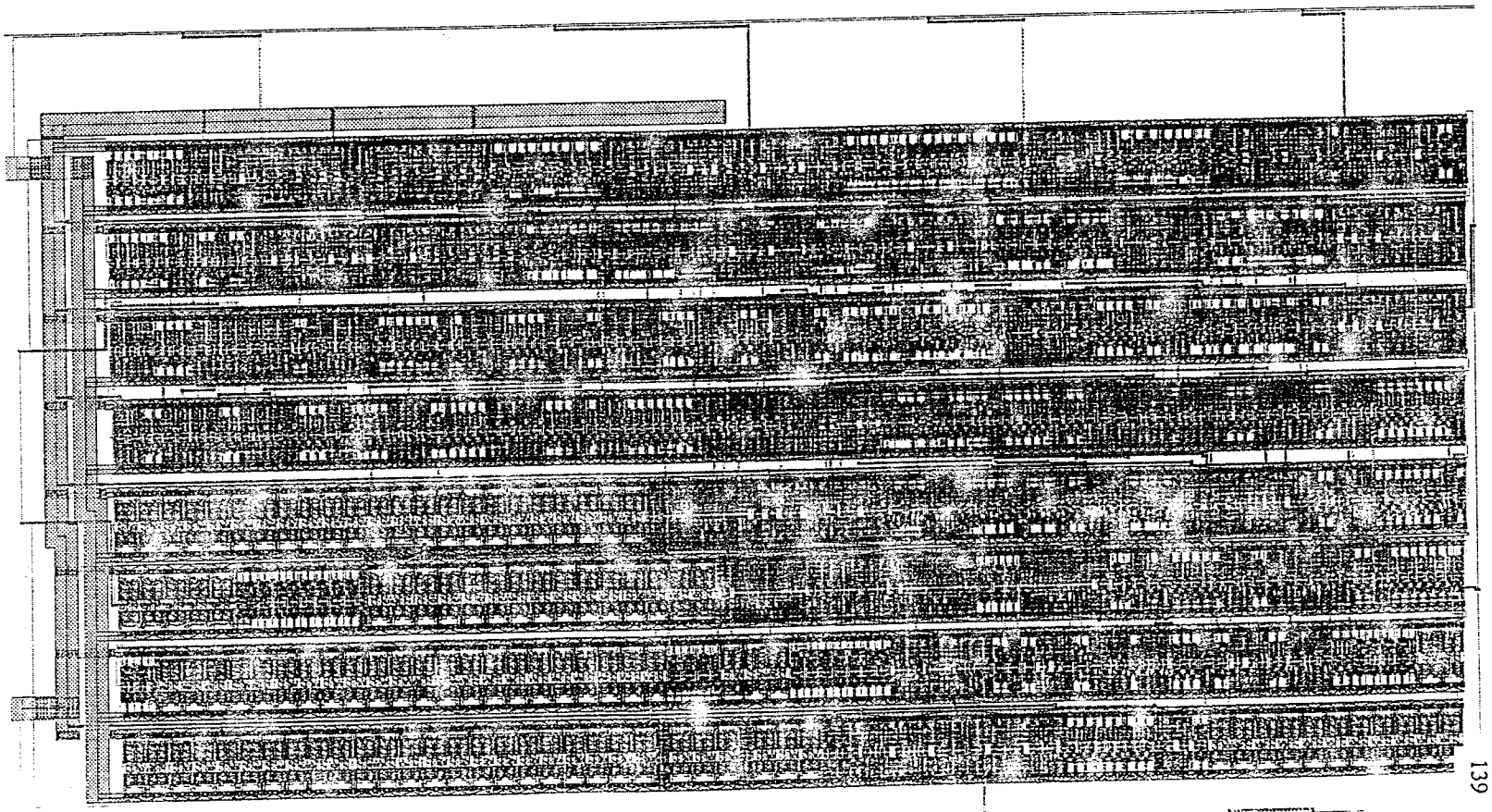


B-5 Schematic of Frequency Multiplier (FM)

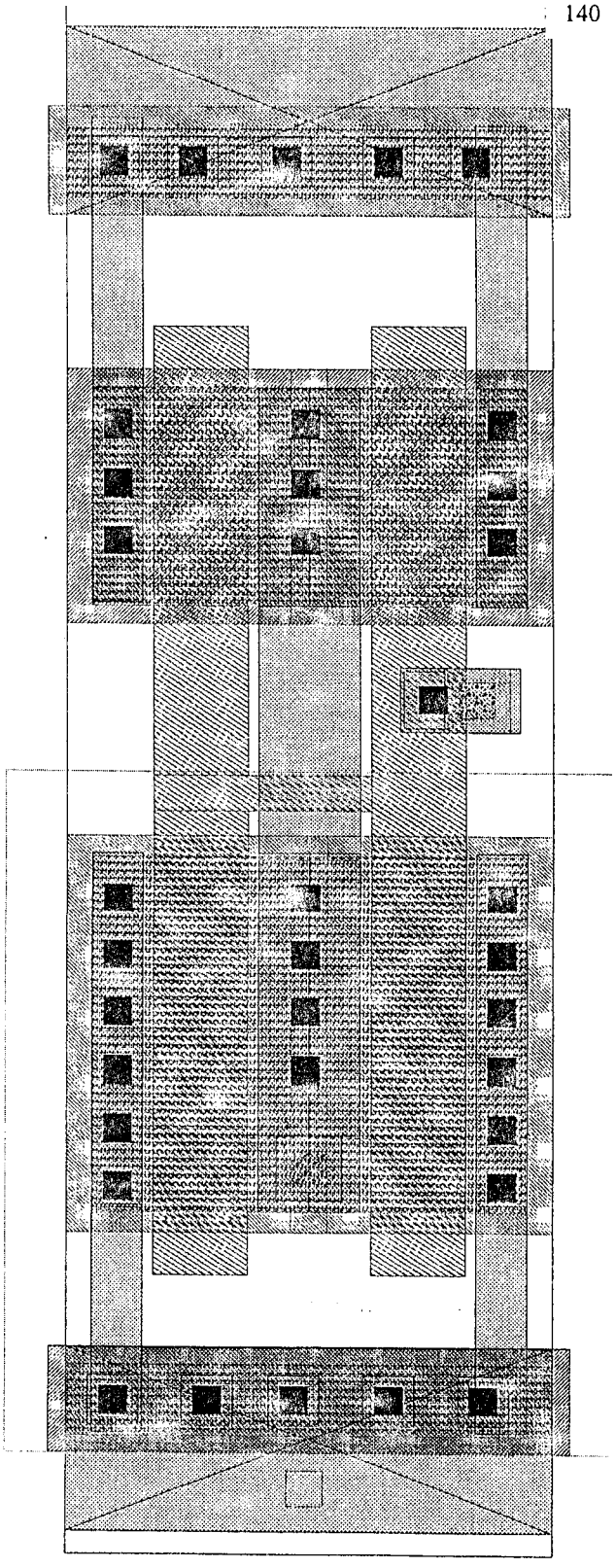
C-1 Whole chip layout for Multi-High-Frequency Synchronous Clock Generator (SCG)



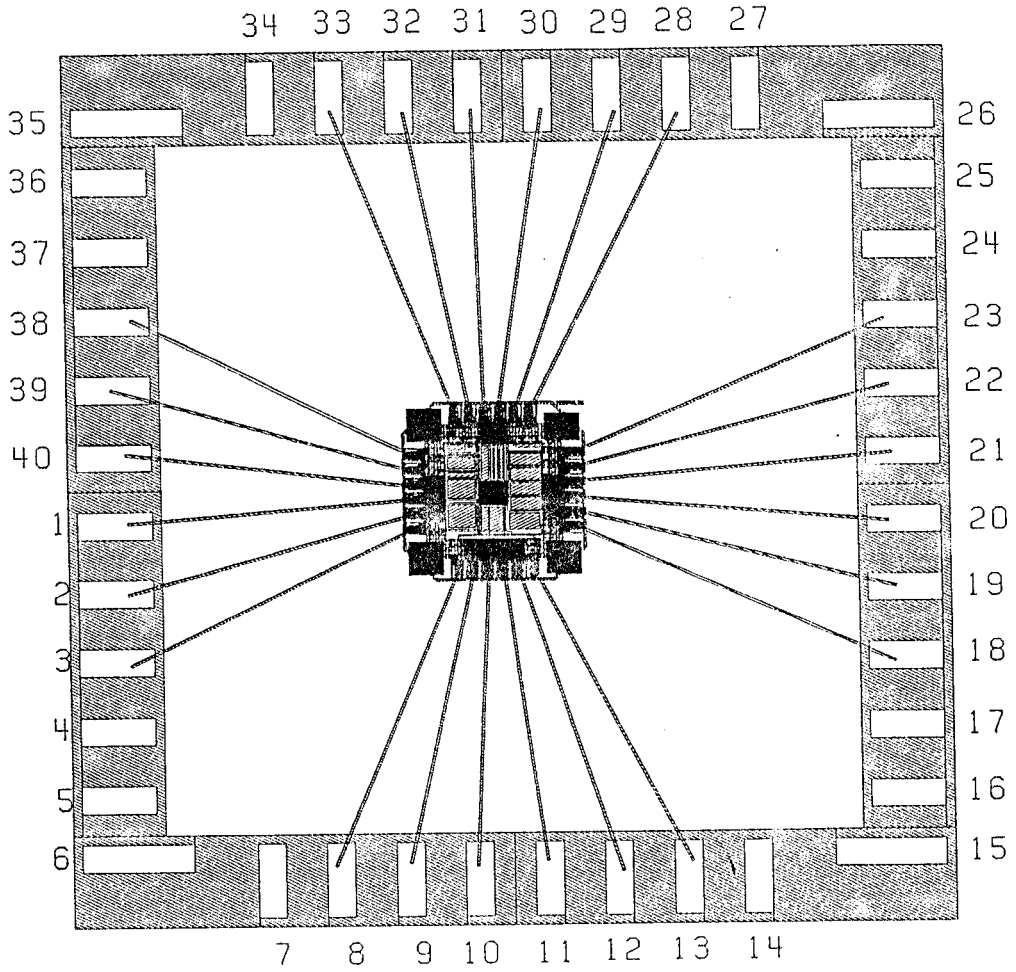
C-2 Layout for SCG core



C-3 Manual layout for bigger size inverter

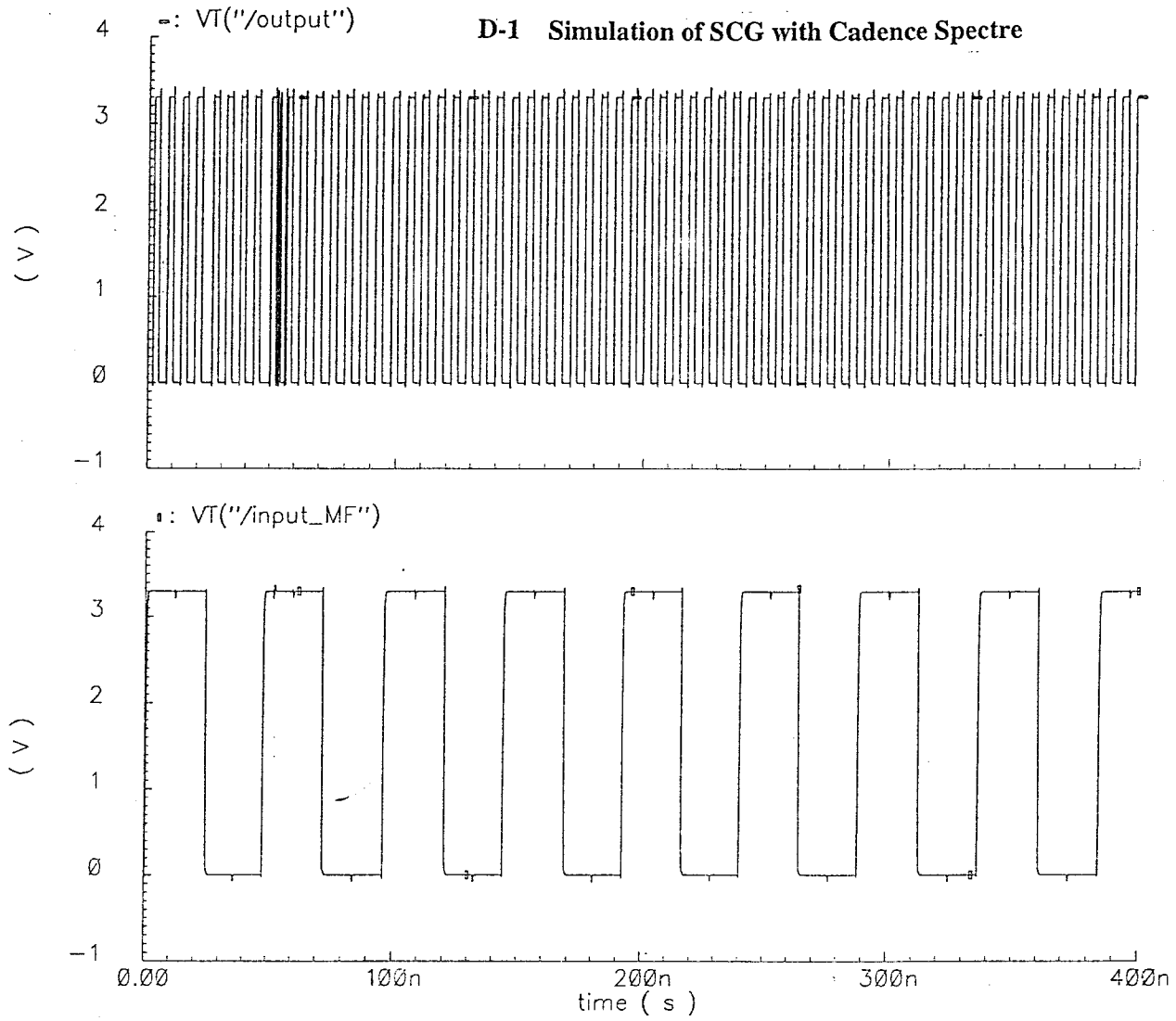


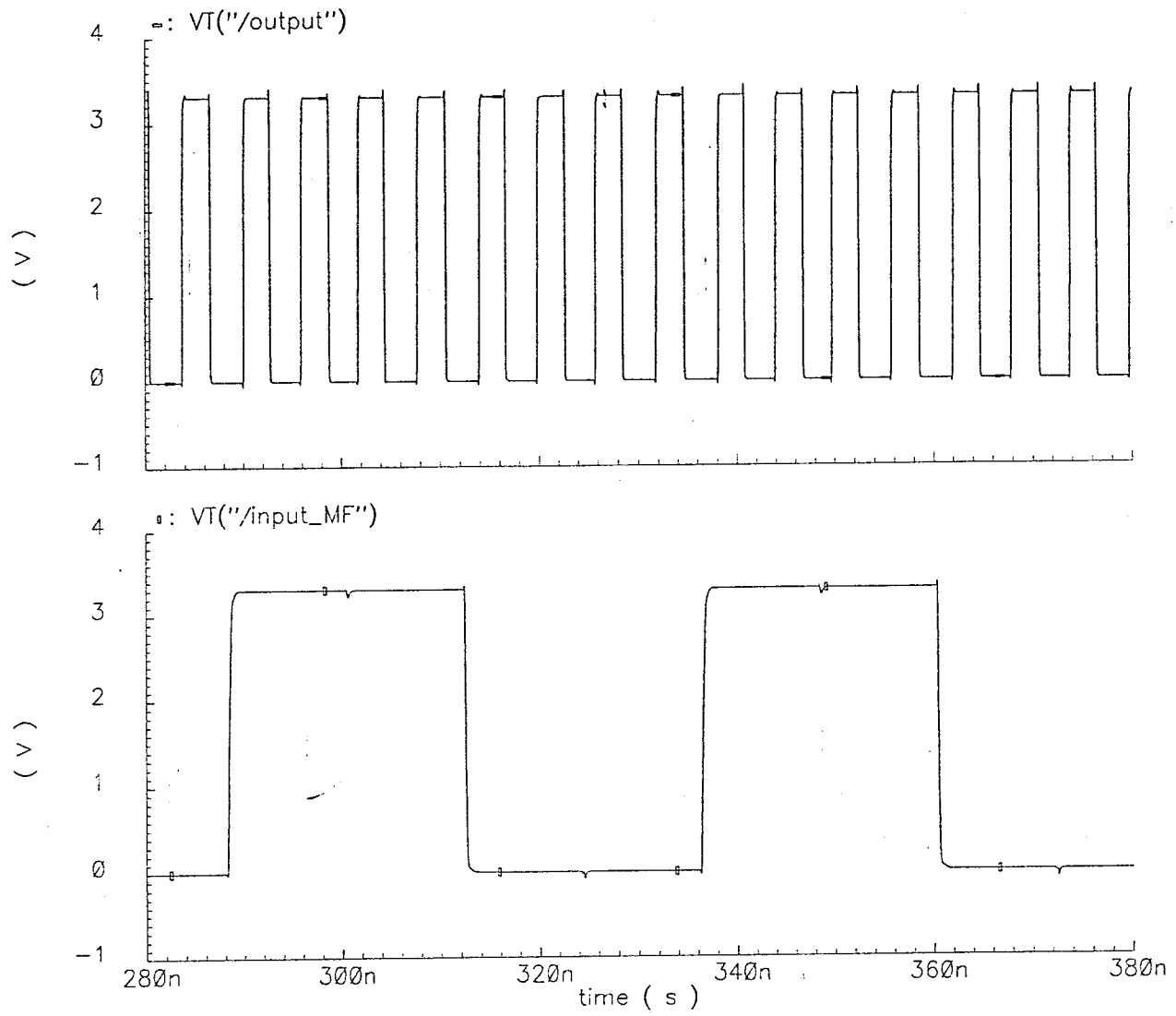
CANADIAN MICROELECTRONICS CO
SOCIETE CANADIENNE DE MICRO-EI

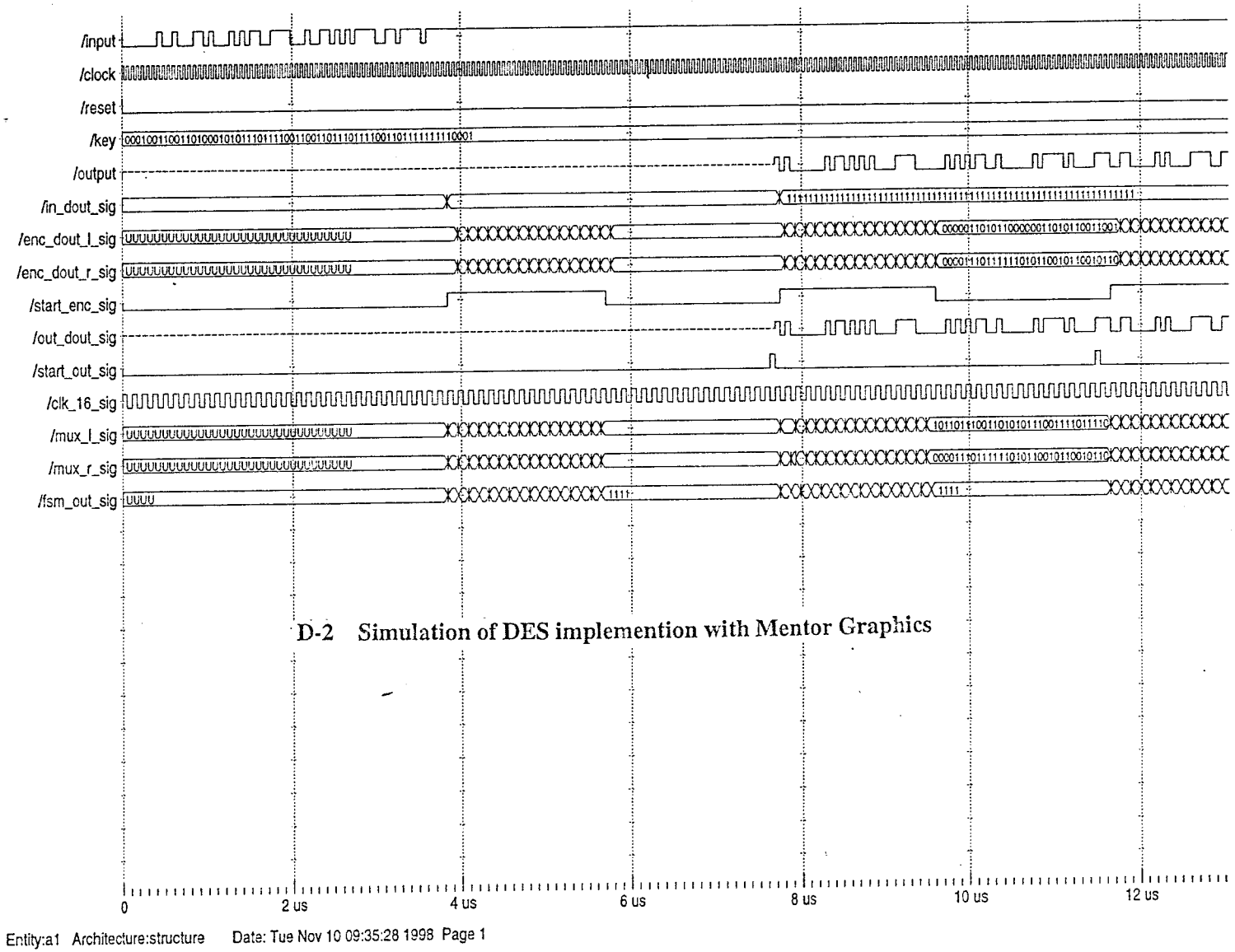


C-4 Bonding diagram for SCG chip package

D-1 Simulation of SCG with Cadence Spectre




: VT("/output")
4
3
2
1
0
−1
(V)
: VT("/input_MF")
4
3
2
1
0
−1
(V)
280n
300n
320n
340n
360n
380n
time (s)



D-2 Simulation of DES implemention with Mentor Graphics

D-3 Timing report of DES implemention with Xilinx xc4028-ex device

```
****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : TOP
Version: 1998.02
Date   : Tue Oct 19 12:19:13 1999
****************************************

Operating Conditions: WCCOM   Library: xprim_4028ex-2
Wire Loading Model Mode: top

Design            Wire Loading Model      Library
------------------------------------------------
TOP                       4028ex-2_avg      xprim_4028ex-2


  Startpoint: U_S/U204 (rising edge-triggered flip-flop clocked by Clock)
  Endpoint: U_S/U210 (rising edge-triggered flip-flop clocked by Clock)
  Path Group: Clock
  Path Type: max

  Point                                                    Incr       Path
  ---------------------------------------------------------------------------
  clock Clock (rise edge)                                  0.00       0.00
  clock network delay (ideal)                              0.00       0.00
  U_S/U204/K (clb_4000)                                    0.00       0.00 r
  U_S/U204/XQ (clb_4000)                                   7.44       7.44 r
  U_S/U_1/gte_2486/B<2> (S_Generater_comp_le_ub_7_1)       0.00       7.44 r
  U_S/U_1/gte_2486/u8/Z (COMP_LE_UBIN_8)                  15.19      22.63 r
  U_S/U_1/gte_2486/Z (S_Generater_comp_le_ub_7_1)          0.00      22.63 r
  U_S/U212/X (clb_4000)                                    4.72      27.35 f
  U_S/U214/X (clb_4000)                                    6.84      34.19 r
  U_S/U210/C4 (clb_4000)                                   0.00      34.19 r
  data arrival time                                                  34.19

  clock Clock (rise edge)                                 50.00      50.00
  clock network delay (ideal)                              0.00      50.00
  U_S/U210/K (clb_4000)                                    0.00      50.00 r
  library setup time                                      -1.74      48.26
  data required time                                                 48.26
  ---------------------------------------------------------------------------
  data required time                                                 48.26
  data arrival time                                                 -34.19
  ---------------------------------------------------------------------------
  slack (MET)                                                        14.07
```

D-4 Mapping report of DES implemention with Xilinx xc4028-ex device

```
                Xilinx Mapping Report File for Design "TOP"
          Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : map -p xc4028ex-2-hq304 -o map.ncd xc4000ex.ngd TOP.pcf
Target Device  : x4028ex
Target Package : hq304
Target Speed   : -2
Mapper Version : xc4000ex -- M1.5.29i
Mapped Date    : Wed Oct 20 10:34:03 1999

Design Summary
--------------
   Number of errors:        0
   Number of warnings:      8
   Number of CLBs:            707 out of  1024   69%
      CLB Flip Flops:     223
      CLB Latches:        128
      4 input LUTs:      1126 (11 used as route-throughs)
      3 input LUTs:       242 (73 used as route-throughs)
   Number of bonded IOBs:      59 out of   256   23%
      IOB Flops:            1
      IOB Latches:          0
   Number of clock IOB pads:    1 out of    12    8%
   Number of BUFGLSs:           1 out of     8   12%
Total equivalent gate count for design: 9812
Additional JTAG gate count for IOBs:    2832

Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Design Attributes
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - Added Logic
Section 7 - Expanded Logic
Section 8 - Signal Cross-Reference
Section 9 - Symbol Cross-Reference
Section 10 - IOB Properties
Section 11 - RPMs
Section 12 - Guide Report
```

```
/*************************************************************
/
/ E-1     C++ codes for DES algorithm
/
*************************************************************/

# include <fstream.h>
# include <stdio.h>
# include <iomanip.h>
# include <stdlib.h>
# include <math.h>

main()
{

     ifstream input("data_key");
     char A[2];
     register int i, j, m, n;
     static unsigned X[64], B[56], C1[28], D1[28], K[49], key[16][48];
     int PC1[56] = {57, 49, 41, 33, 25, 17,  9,
                     1, 58, 50, 42, 34, 26, 18,
                    10,  2, 59, 51, 43, 35, 27,
                    19, 11,  3, 60, 52, 44, 36,
                    63, 55, 47, 39, 31, 23, 15,
                     7, 62, 54, 46, 38, 30, 22,
                    14,  6, 61, 53, 45, 37, 29,
                    21, 13,  5, 28, 20, 12,  4};

   int Z[16] = {1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1};

   int PC2[48] = {14, 17, 11, 24,  1,  5,
                   3, 28, 15,  6, 21, 10,
                  23, 19, 12,  4, 26,  8,
                  16,  7, 27, 20, 13,  2,
                  41, 52, 31, 37, 47, 55,
                  30, 40, 51, 45, 33, 48,
                  44, 49, 39, 56, 34, 53,
                  46, 42, 50, 36, 29, 32};

   for(i=1; i< 65; ++i)
    {
     input.read((char*)&A, sizeof(A));
     X[i] = atoi(A);
    }

   cout << "data_key = " ;
   for( i=1; i<65; ++i)
      {
      cout<< setw(1) << X[i];
      }
      cout<< endl;


 for( i=1; i<57; i++)
     {
        B[i] = X[PC1[i-1]];
     }
```



```
     /*-------------------------------------------------------*/

     for ( m=0; m<16; m++ )
        {

        for ( j =1; j<29-Z[m]; j++ )
             {
                C1[j] = B[j+Z[m]];
             }
             if (Z[m] == 1)
             {
                C1[28] = B[1];

             }
             else if (Z[m] == 2)
             {
                C1[27] = B[1];
                C1[28] = B[2];
             }

             for( i=1; i<29; i++)
             {
             B[i] = C1[i];

             }


      for (j=29; j<57-Z[m]; j++ )
             {
                D1[j] = B[j +Z[m]];
             }
             if (Z[m] == 1)
             {
                D1[56] = B[29];
             }
             else if (Z[m] == 2)
             {
                D1[55] = B[29];
                D1[56] = B[30];
             }


             for( i=29; i<57; i++)
             {
             B[i] = D1[i];

             }

        for( i=1; i<49; i++)
        {
        K[i] = B[PC2[i-1]];
        key[m][i] = K[i];
        }

   }
```

```
        ifstream input1("data_in");
        register unsigned X_in[64], X_IP[64], L[32], R[32], L_32[32],
                          R_48[48], ER[48], S_in[48], S_32[i];
        char A1[2];


      int IP[64]={58, 50, 42, 34, 26, 18, 10, 2,
                  60, 52, 44, 34, 28, 20, 12, 4,
                  62, 54, 46, 38, 30, 22, 14, 6,
                  64, 56, 48, 40, 32, 24, 16, 8,
                  57, 49, 41, 33, 25, 17,  9, 1,
                  59, 51, 43, 35, 27, 19, 11, 3,
                  61, 53, 45, 37, 29, 21, 13, 5,
                  63, 55, 47, 39, 31, 23, 15, 7};

       int IP_I[64] = {40, 8, 48, 16, 56, 24, 64, 32,
                       39, 7, 47, 15, 55, 23, 63, 31,
                       38, 6, 46, 14, 54, 22, 62, 30,
                       37, 5, 45, 13, 53, 21, 61, 29,
                       36, 4, 44, 12, 52, 20, 60, 28,
                       35, 3, 43, 11, 51, 19, 59, 27,
                       34, 2, 42, 10, 50, 18, 58, 26,
                       33, 1, 41,  9, 49, 17, 57, 25};

       int E[48]={32,  1,  2,  3,  4,  5,
                   4,  5,  6,  7,  8,  9,
                   8,  9, 10, 11, 12, 13,
                  12, 13, 14, 15, 16, 17,
                  16, 17, 18, 19, 20, 21,
                  20, 21, 22, 23, 24, 25,
                  24, 25, 26, 27, 28, 29,
                  28, 29, 30, 31, 32,  1};


    cout <<"data_in =  ";
    for(i=0; i<64; ++i)
       {
       input1.read((char*)&A1, sizeof(A1));
       X_in[i] = atoi(A1);
       cout<< X_in[i];
       }
     cout<< endl;

cout <<"/*-------------------------------------------------------*/" <<endl;


     for(i=0; i<64; ++i)
        {
        X_IP[i] = X_in[IP[i]-1];

        }

/*-------------------------------------------------------*/

        cout <<"L0"<< " =          ";
        for(i=0; i<32; ++i)
           {
           L[i] = X_IP[i];
           cout<< L[i];
```



```
           }
         cout<< endl;

         cout <<"L1 = R0"<< " =     ";
         for(i=0; i<32; ++i)
            {
            R[i] = X_IP[i+32];
            cout<< R[i];
            }
         cout << endl;

cout <<"/*-------------------------------------------------------*/" << endl;


for(j=0; j<16; ++j)
  {
        cout << "***********round" <<j+1<< "***********"<< endl;

        for(i=0; i<32; ++i)
           {
            R_48[i]= R[i];
            }

        cout << "E(R"<<j<<")" << "=           ";
        for(i=0; i<48; ++i)
           {
           ER[i] = R_48[E[i]-1];
           cout<< ER[i];
           }
        cout<<endl;


       cout << "K" << j+1 << " =             ";
       for( i=1; i<49; i++)
          {
          cout << key[j][i];
          }
       cout<< endl;

        cout << "E(R"<<j<< ") xor K"<< j+1 << " =   ";
        for(i=0; i<48; ++i)
           {
           S_in[i] = ER[i]^key[j][i+1];
           cout<<S_in[i];
           }
        cout<<endl;

//------- S_BOX-----------------------------------------

       int X1=0, X2=0, X3=0, X4=0, X5=0, X6=0, X7=0, X8=0;
       int Y1=0, Y2=0, Y3=0, Y4=0, Y5=0, Y6=0, Y7=0, Y8=0;
       int LL[32];
       int S1[6], S2[6], S3[6], S4[6], S5[6], S6[6], S7[6], S8[6];
       int C1[2], C2[2], C3[2], C4[2], C5[2], C6[2], C7[2], C8[2];
       int D1[4], D2[4], D3[4], D4[4], D5[4], D6[4], D7[4], D8[4];
       int sequence[32] = {16,  7, 20, 21, 29, 12, 28, 17,
                           1, 15, 23, 26,  5, 18, 31, 10,
                           2,  8, 24, 14, 32, 27,  3,  9,
                           19, 13, 30, 6, 22, 11,  4, 25};
```

```
int E[2] = {0,5};
int F[4] = {1,2,3,4};
char temp[32], P[32];
char * W1, * W2, * W3, * W4, * W5, * W6, * W7, * W8;
char * ROM1[4][16] =
{"1110","0100","1101","0001","0010","1111","1011","1000",
 "0011","1010","0110","1100","0101","1001","0000","0111",
 "0000","1111","0111","0100","1110","0010","1101","0001",
 "1100","0110","1100","1011","1001","0101","0011","1000",
 "0100","0001","1110","1000","1101","0110","0010","1011",
 "1111","1100","1001","0111","0011","1010","0101","0000",
 "1111","1000","1000","0010","0100","1001","0001","0111",
 "0101","1011","0011","1110","1010","0000","0110","1101"};

char * ROM2[4][16] =
{"1111", "0001", "1000", "1110", "0110", "1011", "0011", "0100",
 "1001", "0111", "0010", "1101", "1100", "0000", "0101", "1010",
 "0011", "1101", "0100", "0111", "1111", "0010", "1000", "1110",

 "1100", "0000", "0001", "1010", "0110", "1001", "1011", "0101",
 "0000", "1110", "0111", "1011", "1010", "0100", "1101", "0001",
 "0101", "1000", "1100", "0110", "1001", "0011", "0010", "1111",
 "1101", "1000", "1010", "0001", "0011", "1111", "0100", "0010",
 "1011", "0110", "0111", "1100", "0000", "0101", "1110", "1001"};

  char * ROM3[4][16] =
 {"1010", "0000", "1001", "1110", "0110", "0011", "1111", "0101",
  "0001", "1101", "1100", "0111", "1011", "0100", "0010", "1000",
  "1101", "0111", "0000", "1001", "0011", "0100", "0110", "1010",
  "0010", "1000", "0101", "1110", "1100", "1011", "1111", "0001",
  "1101", "0110", "0100", "1001", "1000", "1111", "0011", "0000",
  "1011", "0001", "0011", "1100", "0101", "1010", "1110", "0111",
  "0001", "1010", "1101", "0000", "0110", "1001", "1000", "0111",
  "1100", "1111", "1110", "0011", "1011", "0101", "0010", "1100"};

  char * ROM4[4][16] =
  {"0111", "1101", "1110", "0011", "0000", "0110", "1001", "1010",
   "0001", "0010", "1000", "0101", "1011", "1100", "0100", "1111",
   "1101", "1000", "1011", "0101", "0110", "1111", "0000", "0011",
   "0100", "0111", "0010", "1100", "0001", "1010", "1110", "1001",
   "1010", "0110", "1001", "0000", "1100", "1011", "0111", "1101",
   "1111", "0001", "0011", "1100", "0101", "0110", "1000", "0100",
   "0011", "1111", "0000", "0110", "1010", "0001", "1101", "1000",
   "1001", "0100", "0101", "1011", "1100", "0111", "0010", "1110"};

  char * ROM5[4][16] =
  {"0010", "1100", "0100", "0001", "0111", "1010", "1011", "0110",
   "1000", "0101", "0011", "1111", "1101", "0000", "1110", "1001",
   "1110", "1011", "0010", "1100", "0100", "0111", "1101", "0001",
   "0101", "0000", "1111", "1010", "0011", "1001", "1000", "0110",
   "0100", "0010", "0001", "1011", "1010", "1101", "0111", "1000",
   "1111", "1001", "1100", "0101", "0110", "0011", "0000", "1110",
   "1011", "1000", "1100", "0111", "0001", "1110", "0010", "1101",
   "0110", "1111", "0000", "1001", "1010", "0100", "0101", "0011"};

  char * ROM6[4][16] =
  {"1100", "0001", "1010", "1111", "1001", "0010", "0110", "1000",
   "0000", "1101", "0011", "0100", "1110", "0111", "0101", "1011",
   "1010", "1111", "0100", "0010", "0111", "1100", "1001", "0101",
```



```
 "0110", "0001", "1101", "1110", "0000", "1011", "0011", "1000",
 "1001", "1110", "1111", "0101", "0010", "1000", "1100", "0011",
 "0111", "0000", "0100", "1010", "0001", "1101", "1011", "0110",
 "0100", "0011", "0010", "1100", "1001", "1101", "1111", "1010",
 "1011", "1110", "0001", "0111", "0110", "0000", "1000", "1101"};

char * ROM7[4][16] =
{"0100", "1011", "0010", "1110", "1111", "0000", "1000", "1101",
 "0011", "1100", "1001", "0111", "0101", "1010", "0110", "0001",
 "1101", "0000", "1011", "0111", "0100", "1001", "0001", "1010",
 "1110", "0011", "0101", "1100", "0010", "1111", "1000", "0110",
 "0001", "0100", "1011", "1101", "1100", "0011", "0111", "1110",
 "1010", "1111", "0110", "1000", "0000", "0101", "1001", "0010",
 "0110", "1011", "1101", "1000", "0001", "0100", "1010", "0111",
 "1001", "0101", "0000", "1111", "1110", "0010", "0011", "1100"};

char * ROM8[4][16] =
{"1101", "0010", "1000", "0100", "0110", "1111", "1011", "0001",
 "1010", "1001", "0011", "1110", "0101", "0000", "1100", "0111",
 "0001", "1111", "1101", "1000", "1010", "0011", "0111", "0100",
 "1100", "0101", "0110", "1001", "0000", "1110", "1001", "0010",
 "0111", "1011", "0100", "0001", "1001", "1100", "1110", "0010",
 "0000", "0110", "1010", "1101", "1111", "0011", "0101", "1000",
 "0010", "0001", "1110", "0111", "0100", "1010", "1000", "1101",
 "1111", "1100", "1001", "0000", "0011", "0101", "0110", "1101"};

      for( i=0; i<6; ++i)
            {
            S1[i] = S_in[i];
            }

         for( i=0; i<6; ++i)
            {
            S2[i] = S_in[i+6];
            }

       for( i=0; i<6; ++i)
            {
            S3[i] = S_in[i+12];
            }

         for( i=0; i<6; ++i)
            {
            S4[i] = S_in[i+18];
            }

         for( i=0; i<6; ++i)
            {
            S5[i] = S_in[i+24];

            }

         for( i=0; i<6; ++i)
            {
            S6[i] = S_in[i+30];

            }

         for( i=0; i<6; ++i)
```

```
                    {
                    S7[i] = S_in[i+36];

                    }

                  for( i=0; i<6; ++i)
                    {
                    S8[i] = S_in[i+42];

                    }

//1-------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C1[i]= S1[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D1[i]= S1[F[i]];

        }

  for( i=0; i<2; i++)
     {
  X1 = X1 + C1[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y1 = Y1 + D1[3-i]*int(pow(2, i));
     }

  W1 = ROM1[X1][Y1];

//2-------------------------------------------------------------
       for( i=0; i<2; i++)
        {
        C2[i]= S2[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D2[i]= S2[F[i]];

        }

  for( i=0; i<2; i++)
     {
  X2 = X2 + C2[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
```



```
  Y2 = Y2 + D2[3-i]*int(pow(2, i));
     }

  W2 = ROM2[X2][Y2];

//3-------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C3[i]= S3[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D3[i]= S3[F[i]];

        }

  for( i=0; i<2; i++)
     {
  X3 = X3 + C3[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y3 = Y3 + D3[3-i]*int(pow(2, i));
     }

  W3 = ROM3[X3][Y3];

//4-------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C4[i]= S4[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D4[i]= S4[F[i]];

        }

  for( i=0; i<2; i++)
     {
  X4 = X4 + C4[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y4 = Y4 + D4[3-i]*int(pow(2, i));
     }
```

```
  W4 = ROM4[X4][Y4];

//5-------------------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C5[i]= S5[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D5[i]= S5[F[i]];

        }
  for( i=0; i<2; i++)
     {
  X5 = X5 + C5[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y5 = Y5 + D5[3-i]*int(pow(2, i));
     }


  W5 = ROM5[X5][Y5];
//6-------------------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C6[i]= S6[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D6[i]= S6[F[i]];

        }
  for( i=0; i<2; i++)
     {
  X6 = X6 + C6[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y6 = Y6 + D6[3-i]*int(pow(2, i));
     }
```



```
  W6 = ROM6[X6][Y6];
//7-------------------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C7[i]= S7[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D7[i]= S7[F[i]];

        }

  for( i=0; i<2; i++)
     {
  X7 = X7 + C7[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y7 = Y7 + D7[3-i]*int(pow(2, i));
     }

  W7 = ROM7[X7][Y7];

//8-------------------------------------------------------------------------

       for( i=0; i<2; i++)
        {
        C8[i]= S8[E[i]];

        }

        for( i=0; i<4; i++)
        {
        D8[i]= S8[F[i]];

        }

  for( i=0; i<2; i++)
     {
  X8 = X8 + C8[1-i]*int(pow(2, i));
     }

  for( i=0; i<4; i++)
     {
  Y8 = Y8 + D8[3-i]*int(pow(2, i));
     }

  W8 = ROM8[X8][Y8];

//-------------------------------------------------------------------------
```

```
    for( i=0; i<4; i++)
       {
       P[i] = W1[i];
       }

    for( i=4; i<8; i++)
       {
       P[i] = W2[i- 4];
       }

    for( i=8; i<12; i++)
       {
       P[i] = W3[i-8];
       }

    for( i=12; i<16; i++)
       {
       P[i] = W4[i-12];
       }

    for( i=16; i<20; i++)
       {
       P[i] = W5[i-16];
       }

    for( i=20; i<24; i++)
       {
       P[i] = W6[i-20];
       }

    for( i=24; i<28; i++)
       {
       P[i] = W7[i-24];
       }

    for( i=28; i<32; i++)
       {
       P[i] = W8[i-28];
       }

    cout << "S_box outputs =  ";
    for( i=0; i<32; i++)
       {
       cout<<P[i];
       }
       cout << endl;
//-----------------------end_S_box-----------------------------

     for (i=0; i<32; i++)
      {
      temp[i] = P[sequence[i]-1];
      }
      cout <<"f(R"<<j<<", K"<<j+1<<") =       "<<temp<<endl;

      char qqq[2];
      qqq[1]='\0';

          for(i=0; i<32; ++i)
             {
             qqq[0]=temp[i];
```



```
             LL[i] = L[i]^ atoi(qqq);
             }

          cout << "L"<<j+2<< " = " <<"R"<<j+1<< " =          ";
           for(i=0; i<32; ++i)
              {
              L[i] = R[i];
              R[i] = LL[i];
              cout << R[i];
              }
           cout << endl;

cout <<"/*-------------------------------------------------------*/"<< endl;


  }
  int data_out [64];
        cout << endl;
        cout << "Ciphertext is :" << endl;
        cout << endl;
        for(i=0; i<32; ++i)
           {
           data_out[i] = R[i];
           }
         for(i=32; i<64; ++i)
           {
           data_out[i] = L[i-32] ;
           }
         for(i=0; i<64; ++i)
           {
           cout << data_out[IP_1[i]-1];
           }
         cout <<endl;
cout <<endl;
cout <<"/*-------------------------------------------------------*/"<< endl;


}
```